

从基本原则、惯用法、语法、库、设计模式、内部机制、开发工具和性能优化8方面深入探讨
编写高质量Python代码的技巧、禁忌和最佳实践

Writing Solid Python Code
91 Suggestions to Improve Your Python Program

编写高质量代码

改善Python程序的91个建议

张颖 赖勇浩 著



机械工业出版社
China Machine Press

(Effective系列丛书)

编写高质量代码：改善Python程序的91个建议

张颖 赖勇浩 著

ISBN: 978-7-111-46704-5

本书纸版由机械工业出版社于2014年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目录

Preface前言

为什么要写这本书

读者对象

如何阅读本书

勘误和支持

致谢

第1章 引论

建议1: 理解Pythonic概念

建议2: 编写Pythonic代码

建议3: 理解Python与C语言的不同之处

建议4: 在代码中适当添加注释

建议5: 通过适当添加空行使代码布局更为优雅、合理

建议6: 编写函数的4个原则

建议7: 将常量集中到一个文件

第2章 编程惯用法

建议8: 利用assert语句来发现问题

建议9: 数据交换值的时候不推荐使用中间变量

建议10: 充分利用Lazy evaluation的特性

建议11: 理解枚举替代实现的缺陷

建议12: 不推荐使用type来进行类型检查

建议13: 尽量转换为浮点类型后再做除法

建议14: 警惕eval()的安全漏洞

建议15: 使用enumerate()获取序列迭代的索引和值

建议16: 分清==与is的适用场景

建议17: 考虑兼容性, 尽可能使用Unicode

建议18: 构建合理的包层次来管理module

第3章 基础语法

- 建议19: 有节制地使用`from...import`语句
- 建议20: 优先使用`absolute import`来导入模块
- 建议21: `i+=1`不等于`++i`
- 建议22: 使用`with`自动关闭资源
- 建议23: 使用`else`子句简化循环（异常处理）
- 建议24: 遵循异常处理的几点基本原则
- 建议25: 避免`finally`中可能发生的陷阱
- 建议26: 深入理解`None`，正确判断对象是否为空
- 建议27: 连接字符串应优先使用`join`而不是`+`
- 建议28: 格式化字符串时尽量使用`.format`方式而不是`%`
- 建议29: 区别对待可变对象和不可变对象
- 建议30: `[]`、`()`和`{}`：一致的容器初始化形式
- 建议31: 记住函数传参既不是传值也不是传引用
- 建议32: 警惕默认参数潜在的问题
- 建议33: 慎用变长参数
- 建议34: 深入理解`str()`和`repr()`的区别
- 建议35: 分清`staticmethod`和`classmethod`的适用场景

第4章 库

- 建议36: 掌握字符串的基本用法
- 建议37: 按需选择`sort()`或者`sorted()`
- 建议38: 使用`copy`模块深拷贝对象
- 建议39: 使用`Counter`进行计数统计
- 建议40: 深入掌握`ConfigParser`
- 建议41: 使用`argparse`处理命令行参数
- 建议42: 使用`pandas`处理大型CSV文件
- 建议43: 一般情况使用`ElementTree`解析XML
- 建议44: 理解模块`pickle`优劣

建议45: 序列化的另一个不错的选择——JSON

建议46: 使用`traceback`获取栈信息

建议47: 使用`logging`记录日志信息

建议48: 使用`threading`模块编写多线程程序

建议49: 使用`Queue`使多线程编程更安全

第5章 设计模式

建议50: 利用模块实现单例模式

建议51: 用`mixin`模式让程序更加灵活

建议52: 用发布订阅模式实现松耦合

建议53: 用状态模式美化代码

第6章 内部机制

建议54: 理解`built-in objects`

建议55: `__init__()`不是构造方法

建议56: 理解名字查找机制

建议57: 为什么需要`self`参数

建议58: 理解MRO与多继承

建议59: 理解描述符机制

建议60: 区别`__getattr__()`和`__getattribute__()`方法

建议61: 使用更为安全的`property`

建议62: 掌握`metaclass`

建议63: 熟悉Python对象协议

建议64: 利用操作符重载实现中缀语法

建议65: 熟悉 Python 的迭代器协议

建议66: 熟悉 Python 的生成器

建议67: 基于生成器的协程及`greenlet`

建议68: 理解GIL的局限性

建议69: 对象的管理与垃圾回收

第7章 使用工具辅助项目开发

- 建议70: 从PyPI安装包
- 建议71: 使用pip和yolk安装、管理包
- 建议72: 做paster创建包
- 建议73: 理解单元测试概念
- 建议74: 为包编写单元测试
- 建议75: 利用测试驱动开发提高代码的可测性
- 建议76: 使用Pylint检查代码风格
- 建议77: 进行高效的代码审查
- 建议78: 将包发布到PyPI

第8章 性能剖析与优化

- 建议79: 了解代码优化的基本原则
- 建议80: 借助性能优化工具
- 建议81: 利用cProfile定位性能瓶颈
- 建议82: 使用memory_profiler 和 objgraph 剖析内存使用
- 建议83: 努力降低算法复杂度
- 建议84: 掌握循环优化的基本技巧
- 建议85: 使用生成器提高效率
- 建议86: 使用不同的数据结构优化性能
- 建议87: 充分利用set的优势
- 建议88: 使用multiprocessing克服GIL的缺陷
- 建议89: 使用线程池提高效率
- 建议90: 使用C/C++模块扩展提高性能
- 建议91: 使用 Cython 编写扩展模块

Preface前言

为什么要写这本书

当这本书的写作接近尾声的时候，回过头来看看这一年多的写作历程，不由得心生感叹，这是一个痛并快乐着的过程。不必说牺牲了多少个周末，也不必计算多少个夜晚伏案写作，单是克服写作过程中因疲劳而迸发出来的彷徨、犹豫和动摇等情绪都觉得是件不容易的事情。但不管怎么说，这最终是个沉淀和收获的过程，写作的同时我也和读者们一样在进步。为什么要写这本书？可以说是机缘巧合。机械工业出版社的杨福川老师联系到我，说他们打算策划一本关于高质量Python编程方面的书籍，问我有没有兴趣加入。实话实说，最开始我是持否定态度的，一则因为业余时间实在有限，无法保证我“工作和生活要平衡”的理念；二则觉得自己水平有限，在学习Python的道路上我和千千万万读者一样，只是一个普通的“朝圣者”，我也有迷惑不解的时候，在没有修炼到大彻大悟之前拿什么来给人传道授业？是赖勇浩老师的加入给我注入了一针强心剂，他丰富的Python项目经验以及长期活跃于Python社区所积累下来的名望无形中给了我一份信心。杨老师的鼓励和支持也更加坚定了我的态度，经过反复考虑和调整自己的心态，最终我决定和赖老师一起完成这本书。因为我也经历过从零开始的Python学习过程，我也遇到过各种困惑，经历过不同的曲折，这些可能也正是每一个学习Python的人从最初到进阶这一过程中都会遇到的问题。抱着分享自己在学习和工作中所积累的一点微薄经验的心态，我开始了本书的写作之旅。这个过程也被我当作是对自己学过的知识的一种梳理。如果与此同时，还能够给读者带来一些启示和思索，那将是这本书所能带给我的最大收获了。

读者对象

- 有一定的Python基础，希望通过项目最佳实践来提升自己的相关Python人员。
- 希望进一步掌握Python相关内部机制的技术人员。
- 希望写出更高质量、更Pythonic代码的编程人员。
- 开设相关课程的大专院校师生。

如何阅读本书

首先需要注意的是，本书并不是入门级的语法介绍类的书籍，因此在阅读本书之前假定你已经掌握了最基础的Python语法。如果没有，也没有关系，你可以先找一本最简单的介绍Python语法的书籍看看，尝试写几个Python小程序之后再次阅读本书。

本书分为8章，主要从编程惯用法、基础语法、库、设计模式、内部机制、开发工具、性能剖析与优化等方面解读如何编写高质量的Python程序。每个章节的内容都以建议的形式呈现，这些建议或源于实际项目应用经验，或源于对Python本质的理解和探讨，或源于社区推荐的做法。它们能够帮助读者快速完成从入门到进阶的这个过程。

由于各个章节相对独立，因此无须花费整段的时间从头开始阅读。你可以在空闲的时候选取任意感兴趣的小节阅读。为了减轻读者负担，本书代码尽量保持完整，阅读过程中无须额外下载其他相关代码。

勘误和支持

由于作者的水平有限，加之编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如果你在阅读过程中遇到任何问题或者发现任何错误，欢迎发送邮件至邮箱

highqualitypython@163.com，我们会尽量一一解答直到你满意。期待能够得到你的真挚反馈。

致谢

首先要感谢机械工业出版社华章公司的杨福川老师，因为有了你的鼓励才使我有勇气开始这本书。还要感谢机械工业出版社的孙海亮编辑，在这一年多的时间中始终支持我的写作，是你的鼓励和帮助引导我顺利完成全部书稿。当然也要感谢我的搭档赖老师，和你合作是一件非常愉快的事情，也让我收获颇多。

其次要感谢我的家人，是你们的宽容、支持和理解给了我完成本书的动力，也是你们无微不至的照顾让我不必为生活中的琐事烦心，从而能全身心地投入到写作中去。

最后，我想提前感谢一下本书的读者，谢谢你们能够选择阅读这本书，这将是作为作者的我们最大的荣幸。

谨以此书献给所有热爱Python的朋友们！

张颖

第1章 引论

“罗马不是一天建成的”，编写代码水平的提升也不可能一蹴而就，通过一点一滴的积累，才能达成从量变到质变的飞跃。这种积累可以从很多方面取得，如一些语言层面的使用技巧、常见的注意事项、编程风格等。本章主要探讨Python中常见的编程准则，从而帮助读者进一步理解Pythonic的本质。本章内容包括如何编写Pythonic代码、在实际应用中需要注意的一些事项和值得提倡的一些做法。希望读者通过对本章的学习，可以在实际应用Pythonic的过程中得到启发和帮助。

建议1：理解Pythonic概念

什么是Pythonic？这是很难定义的，这就是为什么大家无法通过搜索引擎找到准确答案的原因。但很难定义的概念绝非意味着其定义没有价值，尤其不能否定它对编写优美的Python代码的指导作用。

对于Pythonic的概念，众人各有自己的看法，但大家心目之中都认同一个更具体的指南，那就是Tim Peters的《The Zen of Python》（Python之禅）。在这一充满着禅意的诗篇中，有几点非常深入人心：

- 美胜丑，显胜隐，简胜杂，杂胜乱，平胜陡，疏胜密。

- 找到简单问题的一个方法，最好是唯一的方法（正确的解决之道）。

- 难以解释的实现，源自不好的主意；如有非常棒的主意，它的实现肯定易于解释。

不仅这几条，其实《Python之禅》中的每一句都可作为编程的信条。是的，不仅是作为编写Python代码的信条，以它为信条编写出的其他语言的代码也会非常漂亮。

（1）Pythonic的定义

遵循Pythonic的代码，看起来就像是伪代码。其实，所有的伪代码都可以轻易地转换为可执行的Python代码。比如在Wikipedia的快速排序^[1]条目中有如下伪代码：

```
function quicksort('array')
  if length('array')
    ≤ 1      return 'array' // an array of zero or one elements is already sorted
  select and remove a pivot element 'pivot' from 'array' // see 'Choice of
pivot' below
  create empty lists 'less' and 'greater'
  for each 'x' in 'array'
    if 'x'
    ≤ 'pivot' then append 'x' to 'less'
    else append 'x' to 'greater'
  return concatenate(quicksort('less'), list('pivot'), quicksort('greater'))
    // two recursive calls
```

实际上，它可以转化为以下同等行数的可以执行的Python代码：

```
def quicksort(array):
    less = []; greater = []
    if len(array) <= 1:
        return array
    pivot = array.pop()
    for x in array:
        if x <= pivot: less.append(x)
        else: greater.append(x)
    return quicksort(less)+[pivot]+quicksort(greater)
```

看，行数一样的Python代码甚至可读性比伪代码还要好吧？但它真的可以运行，结果如下：

```
>>>quicksort([9,8,4,5,32,64,2,1,0,10,19,27])
[0,1,2,4,5,8,9,10,19,27,32,64]
```

所以，综合这个例子来说，**Pythonic**也许可以定义为：充分体现Python自身特色的代码风格。接下来就看看这样的代码风格在实际中是如何体现的。

(2) 代码风格

对于风格，光说是没有用的，最好是通过例子来看看，因为例子看得见，会显得更真实。下面以语法、库和应用程序为例给大家介

绍。

在语法上，代码风格要充分表现Python自身特色。举个最常见的例子，在其他的语言（如C语言）中，两个变量交换需要如下的代码：

```
int a = 1, b = 2;
int tmp = a;
a = b;
b = tmp;
```

利用Python的packaging/unpackaging机制，Pythonic的代码只需要以下一行：

```
a, b = b, a
```

还有，在遍历一个容器的时候，类似其他编程语言的代码如下：

```
length = len(alist)
i = 0
while i < length:
    do_sth_with(alist[i])
    i += 1
```

而 Pythonic的代码如下：

```
for i in alist:
    do_sth_with(i)
```

灵活地使用迭代器是一种Python风格。又比如，需要安全地关闭文件描述符，可以使用以下with语句：

```
with open(path, 'r') as f:
    do_sth_with(f)
```

通过上述代码的对比，能让大家清晰地认识到Pythonic的一个要求，就是对Python语法本身的充分发挥，写出来的代码带着Python味儿，而不是看着像C语言代码，或者Java代码。

应当追求的是充分利用Python语法，但不应当过分地使用奇技淫巧，比如利用Python的Slice语法，可以写出如下代码：

```
a = [1,2,3,4]
c = 'abcdef'
print a[::-1]
print c[::-1]
```

如果不是同样追求每一个语法细节的“老鸟”，这段代码的作用恐怕不能一眼就看出来。实际上，这个时候更好地体现Pythonic的代码是充分利用Python库里reversed()函数的代码。

```
print list(reversed(a))
print list(reversed(c))
```

(3) 标准库

写Pythonic程序需要对标准库有充分的理解，特别是内置函数和内置数据类型。比如，对于字符串格式化，一般这样写：

```
print 'Hello %s!' % ('Tom',)
```

其实%s是非常影响可读性的，因为数量多了以后，很难清楚哪一个占位符对应哪一个实参。所以相对应的Pythonic代码是这样的：

```
print 'Hello %(name)s!' % {'name': 'Tom'}
```

这样在参数比较多的情况下尤其有用。

```
#
字符串
value = {'greet': 'Hello world', 'language': 'Python'}
print '%(greet)s from %(language)s.' % value
```

%占位符来自于大家的先验知识，其实对于新手而言，有点“莫名其妙”，所以更具有Pythonic风格的代码如下：

```
print '{greet} from {language}.'.format(greet = 'Hello world', language =
'Python')
```

`str.format()`方法非常清晰地表明了这条语句的意图，而且模板的使用也减少了许多不必要的字符，使可读性得到了很大的提升。事实上，`str.format()`也成了Python最为推荐的字符串格式化方法，当然也是最Pythonic的。

(4) Pythonic的库或框架

编写应用程序的时候的要求会更高一些。因为编写应用程序一般需要团队合作，那么可能你编写的那一部分正好是团队的另一成员需要调用的接口，换言之，你可能正在编写库或框架。

程序员利用Pythonic的库或框架能更加容易、更加自然地完成任务。如果用Python编写的库或框架迫使程序员编写累赘的或不推荐的代码，那么可以说它并不Pythonic。现在业内通常认为Flask这个框架是比较Pythonic的，它的一个Hello world级别的用例如下：

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello():
    return "Hello world!"
```

```
if __name__ == "__main__":  
    app.run()
```

稍有编程经验的人都可以通过上例认识到利用Python编程极为容易这一事实。一个Pythonic的框架不会对已经通过惯用法完成的东西重复发明“轮子”，而且它也遵循常用的Python惯例。创建Pythonic的框架极其困难，什么理念更酷、更符合语言习惯对此毫无帮助，事实上这些年来优秀的Python代码的特性也在不断演化。比如现在认为像generators之类的特性尤为Pythonic。

另一个有关新趋势的例子是：Python的包和模块结构日益规范化。现在的库或框架跟随了以下潮流：

- 包和模块的命名采用小写、单数形式，而且短小。
- 包通常仅作为命名空间，如只包含空的__init__.py文件。

[1] <http://en.wikipedia.org/wiki/Quicksort>。

建议2：编写Pythonic代码

如何编写更加Pythonic的代码，与定义什么是Pythonic一样困难。在这里，只能给出一些经验之谈，希望对大家有所帮助。

(1) 要避免劣化代码

与优化代码对应，劣化代码就是一开始写出来就是不合理的代码，比如不合适的变量命名等。通常有以下几个值得注意的地方：

1) 避免只用大小写来区分不同的对象。如a是一个数值类型变量，A是String类型，虽然在编码过程中很容易区分二者的含义，但这样做毫无益处，它不会给其他阅读代码的人带来多少便利。

2) 避免使用容易引起混淆的名称。容易引起混淆的名称的使用情形包括：重复使用已经存在于上下文中的变量名来表示不同的类型；误用了内建名称来表示其他含义的名称而使之在当前命名空间被屏蔽；没有构建新的数据类型的情况下使用类似于element、list、dict等作为变量名；使用o（字母O小写的形式，容易与数值0混淆）、l（字母L小写的形式，容易与数值1混淆）等作为变量名。因此推荐变量名与所要解决的问题域一致。有如下两个示例，示例二比示例一更好。

示例一：

```
>>> def funA(list,num):
...     for element in list:
...         if num==element:
...             return True
...         else:
```

```
... pass
...
```

示例二:

```
>>> def find_num(searchList,num):
...     for listValue in searchList:
...         if num==listValue:
...             return True
...         else:
...             pass
... 
```

3) 不要害怕过长的变量名。为了使程序更容易理解和阅读，有的时候长变量名是必要的。不要为了少写几个字母而过分缩写。下例是一个用来保存用户信息的字典结构，变量名`person_info`比`pi`的可读性要强得多。

```
>>> person_info={'name':'Jon','IDCard':'200304','address':'Num203,Monday Road',
'email':'test@gail.com'}
```

(2) 深入认识Python有助于编写Pythonic代码

可以从以下几个方面进行着手:

- 全面掌握Python提供给我们的所有特性，包括语言特性和库特性。其中最好的学习方式应该是通读官方手册中的**Language Reference**和**Library Reference**。掌握了语言特性和库特性，以后许多“惯用法”自然而然就掌握了，写代码的时候，自然会使用常见的、公认的、简短的惯用法来实现预期效果，也使得代码显得尤为Pythonic。

- 随着Python的版本更新、时间的推移，Python语言不断演进，社区不断成长，还需要学习每个Python新版本提供的新特性，以及掌握它的变化趋势。从另一角度来看，一方面Python语言推荐使用大量的

惯用法来完成任务（“完成任务的唯一方法”）；另一方面，社区不断演变的新惯用法反过来又影响了语言的进化，以更好地支持惯用法。比如早年的Pythonista常用dict.has_key()方法来判断字典对象是否包含某个元素，但新版本的Python中提供了in操作符（支持多种容器类型）取代它。改变习惯的阻力很大，而克服这些阻力的唯一方法就是加深对Python的认识，因为在语言支持正确的惯用法之后，非推荐的代码通常执行起来更慢。所以说，不更新知识是不行的。

·深入学习业界公认的比较Pythonic的代码，比如Flask、gevent和requests等。以requests这个通过HTTP（HTTPS）协议获取网络资源的程序库为例，要获取带有Basic Auth的网络资源时，代码如下：

```
import requests
r = requests.get('https://api.github.com', auth=('user', 'pass'))
print r.status_code
print r.headers['content-type']
# -----
# 200
# 'application/json'
```

而使用Python标准库urllib2时，代码就非常复杂，程序员需要了解相当多的关于HTTP协议和Basic Auth的知识才能编程。

```
import urllib2
gh_url = 'https://api.github.com'
req = urllib2.Request(gh_url)
password_manager = urllib2.HTTPPasswordMgrWithDefaultRealm()
password_manager.add_password(None, gh_url, 'user', 'pass')
auth_manager = urllib2.HTTPBasicAuthHandler(password_manager)
opener = urllib2.build_opener(auth_manager)
urllib2.install_opener(opener)
handler = urllib2.urlopen(req)
print handler.getcode()
print handler.headers.getheader('content-type')
# -----
# 200
# 'application/json'
```

看，使用一个Pythonic的程序库可以简化很多工作量！那么深入学习理解类似requests的高质量程序库带给我们的收获应该完全可以预

期：一定是非常大的！

最后，除了修炼内功外，也可以尝试利用工具达到事半功倍的效果。所以接下来介绍风格检查程序**PEP8**。其实一开始**PEP8**是一篇关于**Python**编码风格的指南，它提出了保持代码一致性的细节要求。它至少包括了对代码布局、注释、命名规范等方面的要求，在代码中遵循这些原则，有利于编写**Pythonic**的代码。比如，对代码的换行，不好的风格如下：

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

而**Pythonic**的风格则是这样的：

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

如果要“人肉”检查代码是否符合**PEP8**规范，则比较困难，而且容易跟同僚引发争论。所以Johann C. Rocholl开发了一个应用程序来进行检测，就是应用程序**PEP8**。当然，它是使用**Python**开发的，安装它非常容易。

```
$ pip install -U pep8
```

在自己的**shell**中执行这一命令就可以安装成功了（首先需要安装**pip**）。然后即可简单地用它检测一下自己的代码。

```
$ pep8 --first optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
```

```
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

可以看到上面有许多错误和警告，然后我们按图索骥逐一修复它们就可以了。如果嫌这种报表不够细致，可以考虑使用`--show-source`参数让PEP8显示每一个错误和警告对应的代码。

```
$ pep8 --show-source --show-pep8 testsuite/E40.py
testsuite/E40.py:2:10: E401 multiple imports on one line
import os, sys
      ^
Imports should usually be on separate lines.
Okay: import os\nimport sys
E401: import sys, os
```

看，它甚至可以给出“正确”的写法！除了针对某一个源代码文件以外，它还可以直接检测一个项目的质量，并通过直观的报表给出报告。

```
$ pep8 --statistics -qq Python-2.5/Lib
232      E201 whitespace after '['
599      E202 whitespace before '))'
631      E203 whitespace before ','
842      E211 whitespace before '('
2531     E221 multiple spaces before operator
4473     E301 expected 1 blank line, found 0
4006     E302 expected 2 blank lines, found 1
165      E303 too many blank lines (4)
325      E401 multiple imports on one line
3615     E501 line too long (82 characters)
612      W601 .has_key() is deprecated, use 'in'
1188     W602 deprecated form of raising exception
```

PEP8有优秀的插件架构，可以方便地实现特定风格的检测（例如，有些公司、团队会定义自己的风格）；它生成的报告易于处理，可以很方便地与编辑器集成（例如，实现点击出错信息跳转到相应代码行）。所以这是一个非常有用的工具，它可以提升你对Pythonic的认识，达到编写高质量代码的目的。

PEP8不是唯一的编程规范，事实上，有些公司制定的编程规范也非常有参考意义，比如Google Python Style Guide。同样，PEP8也不是唯一的风格检测程序，类似的应用还有Pychecker、Pylint、Pyflakes等。其中，Pychecker是Google Python Style Guide推荐的工具；Pylint因可以非常方便地通过编辑配置文件实现公司或团队的风格检测而受到许多人的青睐；Pyflakes则因为易于集成到vim中，所以使用的人也非常多。

其实Pythonic的代码，往往是放弃自我风格的代码，而要有“放弃自我风格”的觉悟，是非常困难、非常痛苦的。要突破这种瓶颈，完成自我蜕变，除了需要付出许多精力去学习外，参考更好的书籍进行辅助也是相当有帮助的。目前市面上针对编写“高质量”的Python程序的方法的书籍并不多，本书应是一本比较好的参考资料。作为作者，我们也真心希望自己的一点点经验分享能够对读者有所帮助。

建议3：理解Python与C语言的不同之处

我们都知道，Python底层是用C语言实现的，但切忌用C语言的思维和风格来编写Python代码。对于那些在学习Python之前有其他编程语言（如Java、C#等）经验的程序员来说，尤其重要的是：不要使用之前的编程思想。Python与它们有很多不同，以下从语法方面来进行简单分析。

(1) “缩进”与“{}”

与C、C++、Java等语言使用花括号{}来分隔代码段不同，Python中使用严格的代码缩进方式分隔代码块，空格或者Tab键不再是你心血来潮的时候可以随便敲敲解闷的了，对于Python解释器而言，它们直接关乎代码的语法和逻辑，一不小心就会出现unexpected indent错误。Python的这一特点也曾引起不少争议，强制代码缩进就像一把双刃剑，有利有弊。对于从其他编程语言转过来学习Python的人来说，也许需要一段时间去适应。但不可否认，严格的缩进确实能让代码更加规范、整齐，可读性和可维护性都会更高。避免缩进带来的困扰的方法之一就是养成良好的习惯，统一缩进风格，不要混用Tab键和空格。

(2) '与"

C语言中单引号（'）与双引号（"）有严格的区别，单引号代表一个字符，它实际对应于编译器所采用的字符集中的一个整数值。例如在ASCII码中，'a'与97相对应。而双引号则表示字符串，默认以'\0'结

尾。但在Python中，单引号与双引号没有明显区别，仅仅在输入字符串内容不同时，在使用上存在微小差异。

```
>>> string1 = "He said, \"Hello\""          #
字符串中本身的双引号需要转义
>>> string1
'He said, "Hello"'
>>> string2 = 'He said, "Hello"'            #
字符串本身的双引号不需要转义
>>> string2
'He said, "Hello"'
```

(3) 三元操作符“?:”

三元操作符是if...else的简写方法，语法形式为C ? X: Y，它表示当条件C为True的时候取值X，C为False的时候取值为Y。其简洁的表达形式一直很受欢迎。但在Python 2.5之前并不支持三元操作符。为此，人们想出了不少替代方式，但在特殊情形下存在一些问题，因此很多人对Python语言本身加入该特征也提出了不少建议，最终PEP308被接受，根据提议采用if...else...形式实现条件表达式。C ? X: Y在Python中等价的形式为X if C else Y，即：

```
>>> X=0
>>> Y=-2
>>> print X if X<Y else Y
-2
```

(4) switch...case

Python中没有像C语言那样的switch...case分支语句，不过这不是什么难事，Python中有很多替代的解决方法。假设，用C语言实现的switch...case语句如下：

```
switch(n) {
    case 0:
        printf("You typed zero.\n");
        break;
    case 1:
        printf("You are in top.\n");
        break;
    case 2:
        printf("n is an even number\n");
    default:
        printf("Only single-digit numbers are allowed\n");
        break;
}
```

与以上C语言中switch...case对应的Python实现如下:

```
if n == 0:
    print "You typed zero.\n"
elif n == 1:
    print "You are in top.\n"
elif n == 2:
    print "n is an even number\n"
else:
    print "Only single-digit numbers are allowed\n"
```

或者使用以下跳转表来实现:

```
def f(x):
    return {
        0: "You typed zero.\n",
        1: "You are in top.\n",
        2: "n is an even number\n"
    }.get(n, "Only single-digit numbers are allowed\n")
```

以上只是简单列举了几个Python和其他语言的不同点，事实上，其差异性远远不止这些。但总归一句话：不要被其他语言的思维和习惯困扰，掌握Python的哲学和思维方式才是硬道理。正如前面所说，要舍得抛弃具有自我风格的代码，尽量遵循Pythonic代码的编码规范和风格。

建议4：在代码中适当添加注释

Python中有3种形式的代码注释：块注释、行注释以及文档注释（docstring）。这3种形式的惯用法大概有如下几种：

1) 使用块或者行注释的时候仅仅注释那些复杂的操作、算法，还有可能别人难以理解的技巧或者不够一目了然的代码。

2) 注释和代码隔开一定的距离，同时在块注释之后最好多留几行空白再写代码。下面两行代码显然第一行的阅读性要好。

```
x=x+1          # increace x  by 1
①
x=x+1 #increase x by 1
②
```

3) 给外部可访问的函数和方法（无论是否简单）添加文档注释。注释要清楚地描述方法的功能，并对参数、返回值以及可能发生的异常进行说明，使得外部调用它的人员仅仅看docstring就能正确使用。较为复杂的内部方法也需要进行注释。推荐的函数注释如下：

```
def FuncName(parameter1 , parameter2):
    """Describe what this function does.
       #such as "Find whether the special string is in the queue or not"
       Args:
           parameter1: parameter type, what is this parameter used for.
                       #such as strqueue:string,string queue list for search
           parameter2: parameter type, what is this parameter used for.
                       #such as str:string,string to find
       Returns:
           return type, return value.
           #such as  boolean,sepcial string  found return True,else return False
    """
    function body
    ...
    ...
```

4) 推荐在文件头中包含copyright申明、模块描述等，如有必要，可以考虑加入作者信息以及变更记录。

```
"""
    Licensed Materials - Property of CorpA
    (C) Copyright A Corp. 1999, 2011 All Rights Reserved
    CopyRight statement and purpose...
    -----
    File Name      : comments.py
    Description    : description what the main function of this file
    Author: Author name
    Change Activity:
                    list the change activity and time and author information.
    -----
    """
```

有人说，写代码就像写诗，你见过谁在自己写的诗里加注释吗？这种说法受到许多人的追捧，包括一些Python程序员。但我的看法是，代码跟诗不太一样，需要适当添加注释。注释直接关系到代码的可读性和可维护性，同时它还能够帮助发现错误的藏身之处。因为代码是说明你怎么做的，而好的注释能够说清楚你想做什么，它们相辅相成。但往往有些程序员并不重视它，原因是多方面的，有人觉得程序的实现才是最重要的，至于注释是一件浪费时间的事情；还有的人明明知道注释很重要，可是太懒，不愿意写或者随便应付；也有人重视注释但却不得要领，反而使其成为代码的一种累赘。下面针对以上几个心态举几个实际中常见例子。

示例一：代码即注释（不写注释）。没有注释的代码通常会给他人的阅读和理解带来一定困难，即使是自己写的代码，过一段时间再回头阅读可能也需要一定时间才能理解当初的思路。

```
a=3
n=5
count,sn,tn=1,0,0
while count<=n:
    .    tn+=a
        sn+=tn
        a*=10
        count+=1
print sn
```

示例二：注释与代码重复。注释应该是用来解释代码的功能、原因以及想法的，而不是对代码本身的解释。

```
# coding=utf-8
print "please input number n"
n=input()
print "please input number m"
m=input()
t=n/2                # t
是n
的一半
#
循环，条件为t*m/n
小于n
while(t*m/(n+1) < n):
    t=0.5*m+n/2      #
重新计算t
值
    print t
```

示例三：利用注释语法快速删除代码。对于不再需要的代码，应该将其删除，而不是将其注释掉。即使你担心以后还会用到，版本控制工具也可以让你轻松找回被删除的代码。

```
x=2
y=5
z=3
"""following code is no longer needed since there is a better way
if x>y:t=x;x=y;y=t
if x>z:t=z;z=x;x=t
if y>z:t=y;y=z;z=t
print "the order from small to big is: %d %d %d" % (x,y,z)
"""
order_list=[x,y,z]
order_list.sort()
print order_list
```

其他比较常见的问题还包括：代码不断更新而注释却没有更新；注释比代码本身还复杂烦琐；将别处的注释和代码一起拷贝过来，但上下文的变更导致注释与代码不同步；更有个别人将注释当做自己的娱乐空间从而留下个性特征等，这几种行为都是平时要注意避免的。

建议5：通过适当添加空行使代码布局更为优雅、合理

布局清晰、整洁、优雅的代码能够给阅读它的人带来愉悦感，而且它能帮助开发者之间进行良好的沟通。在一个团队中，保持良好的代码格式需要团队成员在选取一套合适的代码格式规则的基础上遵从和应用。同时它需要每个团队成员树立正确的态度，因为实际工作中有很多开发者抱着这样的想法：“代码能工作才是最重要的”，但往往代码会不断修改，且可读性直接关系到可维护性和可扩展性。因此我们需要端正态度——“代码不是恒定的，只有风格才能延续，能工作的代码和整洁、优雅的风格同样重要”。

为了让读者更加深入地理解代码布局的重要性，我们先来看一个猜数字游戏的示例。下面两段代码，编码完全相同，只是在排版上做了一定修改，你觉得哪个更加容易阅读呢？

示例一：

```
import random
guesses_made = 0
name = raw_input('Hello! What is your name?\n')
number = random.randint(1, 20)
print 'Well, {0}, I am thinking of a number between 1 and 20.'.format(name)
while guesses_made < 6:
    guess = int(raw_input('Take a guess: '))
    guesses_made += 1
    if guess < number:
        print 'Your guess is too low.'
    if guess > number:
        print 'Your guess is too high.'
    if guess == number:
        Break
if guess == number:
    print 'Good job, {0}! You guessed my number in {1} guesses!'.format(name,
guesses_made)
else:
    print 'Nope. The number I was thinking of was {0}'.format(number)
```

示例二:

```
import random
guesses_made = 0
name = raw_input('Hello! What is your name?\n')
number = random.randint(1, 20)
print 'Well, {0}, I am thinking of a number between 1 and 20.'.format(name)
while guesses_made < 6:
    guess = int(raw_input('Take a guess: '))
    guesses_made += 1
    if guess < number: print 'Your guess is too low.'
    if guess > number: print 'Your guess is too high.'
    if guess == number: break
if guess == number: print 'Good job, {0}! You guessed my number in {1} guesses!'.format(name, guesses_made)
else: print 'Nope. The number I was thinking of was {0}'.format(number)
```

看了上面两个例子，相信很多读者都倾向于阅读第一个例子，这就是代码布局和排版带给我们的最直观的感受（注意：本书其他章节的例子为了尽量使代码占用更少的篇幅，采取了第二种排版形式，但在实际项目开发中更推荐第一种）。和其他语言一样，Python代码布局也有一些基本规则可以遵循。

1) 在一组代码表达完一个完整的思路之后，应该用空白行进行间隔。如每个函数之间，导入声明、变量赋值等。通俗点讲就是不要在一段代码中说明几件事。推荐在函数定义或者类定义之间空两行，在类定义与第一个方法之间，或者需要进行语义分割的地方空一行。如示例一在import声明和变量赋值之间插入了空行。但读者需要注意的是：空行是在不隔断代码之间内在联系的基础上插入的，也就是说有关联的代码还是需要保持紧凑、连续。在示例一中，如果你在if和else之间插入空行就显得非常没有必要，就像下面的代码：

```
if guess == number:
    print 'Good job, {0}! You guessed my number in {1} guesses!'.format(name,
guesses_made)

else:
    print 'Nope. The number I was thinking of was {0}'.format(number)
```

2) 尽量保持上下文语义的易理解性。如当一个函数需要调用另一个函数的时候，尽量将它们放在一起，最好调用者在上，被调用者在下。例如下面的代码：

```
def A():  
    B()  
def B():  
    pass
```

3) 避免过长的代码行，每行最好不要超过80个字符。以每屏能够显示完整代码而不需要拖动滚动条为最佳，超过的部分可以用圆括号、方括号和花括号等进行行连接，并且保持行连接的元素垂直对齐。例如下面的代码：

```
>>> x=('this is a very long string'  
... 'it is used for testing line limited characters')  
>>> print x  
this is a very long stringit is used for testing line limited characters  
>>>def Draw_Line(pointX1,pointY1,pointX2=1,pointY2=2,  
    color='green',width=2,style='bold'):
```

4) 不要为了保持水平对齐而使用多余的空格，其实使阅读者尽可能容易地理解代码所要表达的意义更重要。如下列代码的主要目的是赋值，为了可以保持对齐往往会造成“喧宾夺主”。

```
X = 5  
Year = 2013  
name = "Jam"  
d2 = {'spam': 2, 'eggs': 3}
```

同时也不要在一行有多个命令，如不要将X=1;Y=2;直接写在一行中。

5) 空格的使用要能够在需要强调的时候警示读者，在疏松关系的实体间起到分隔作用，而在具有紧密关系的时候不要使用空格。具体

细节如下：

·二元运算符（赋值（=），比较（==，<，>，!=，<>，<=，>=，in，not in，is，is not）、布尔运算（and，or，not））的左右两边应该有空格，如x == 1。

·逗号和分号前不要使用空格。

推荐：if x == 4: print x, y; x, y = y, x
不推荐：if x == 4 : print x , y ; x , y = y , x

·函数名和左括号之间、序列索引操作时序列名和[]之间不需要空格，函数的默认参数两侧不需要空格。

·强调前面的操作符的时候使用空格，如 -2 - 5（在-2和5之间的减号前后需要添加空格）、b*b + a*c（在加号前后需要添加空格）。

建议6：编写函数的4个原则

函数能够带来最大化的代码重用和最小化的代码冗余。精心设计的函数不仅可以提高程序的健壮性，还可以增强可读性、减少维护成本。先来看以下示例代码：

```
def SendContent(ServerAdr, PagePath, StartLine, EndLine, sender,
               receiver, smtpserver, username, password):
    http = httplib.HTTP(ServerAdr)
    http.putrequest('GET', PagePath)
    http.putheader('Accept', 'text/html')
    http.putheader('Accept', 'text/plain')
    http.endheaders()
    httpcode, httpmsg, headers = http.getreply()
    if httpcode != 200: raise "Could not get document: Check URL and Path."
    doc = http.getfile()
    data = doc.read()
    doc.close()
    lstr=data.splitlines()
    j=0
    for i in lstr:
        j=j+1
        if i.strip() == StartLine: slice_start=j #find slice start
        elif i.strip() == EndLine: slice_end=j #find slice end
    subject = "Contented get from the web"
    msg = MIMEText(string.join(lstr[slice_start:slice_end]), 'plain', 'utf-8')
    msg['Subject'] = Header(subject, 'utf-8')
    smtp = smtplib.SMTP()
    smtp.connect(smtpserver)
    smtp.login(username, password)
    smtp.sendmail(sender, receiver, msg.as_string())
    smtp.quit()
```

函数SendContent主要的作用是抓取网页中固定的内容，然后将其发送给用户。代码本身并不复杂，但足以说明一些问题。读者可以先思考一下：到底有什么不是之处？能否进一步改进？怎样才能做到一目了然呢？一般来说函数设计有以下基本原则可以参考：

原则1 函数设计要尽量短小，嵌套层次不宜过深。所谓短小，就是跟前面所提到的一样尽量避免过长函数，因为这样不需要上下拉动滚动条就能获得整体感观，而不是来回翻动屏幕去寻找某个变量或

者某条逻辑判断等。函数中需要用到if、elif、while、for等循环语句的地方，尽量不要嵌套过深，最好能控制在3层以内。相信很多人有过这样的经历：为了弄清楚哪段代码属于内部嵌套，哪段属于中间层次的嵌套，哪段属于更外一层的嵌套所花费的时间比读代码细节所用时间更多。

原则2 函数申明应该做到合理、简单、易于使用。除了函数名能够正确反映其大体功能外，参数的设计也应该简洁明了，参数个数不宜太多。参数太多带来的弊端是：调用者需要花费更多的时间去理解每个参数的意思，测试人员需要花费更多的精力来设计测试用例，以确保参数的组合能够有合理的输出，这使覆盖测试的难度大大增加。因此函数参数设计最好经过深思熟虑。

原则3 函数参数设计应该考虑向下兼容。实际工作中我们可能面临这样的情况：随着需求的变更和版本的升级，在前一个版本中设计的函数可能需要进行一定的修改才能满足这个版本的要求。因此在设计过程中除了着眼当前的需求还得考虑向下兼容。如以下示例：

```
>>> def readfile(filename):
①函数实现的第一版本
...     print "file read completed"
...
>>> readfile("test.txt")
file read completed
>>>
>>> import logging
>>>
>>> def readfile(filename, logger):
②函数实现的第二版本
...     print "file read completed"
...
>>> readfile("test.txt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: readfile() takes exactly 2 arguments (1 given)
```

上面的代码是相同功能的函数不同版本的实现，唯一不同的是在更高级的版本中为了便于内部维护加入了日志处理，但这样的变动却

导致程序中函数调用的接口发生了改变。这并不是最佳设计，更好的方法是通过加入默认参数来避免这种退化，做到向下兼容。上例可以将第一行代码修改为：

```
def readfile(filename, logger=logger.info):
```

原则4 一个函数只做一件事，尽量保证函数语句粒度的一致性。如本节开头所示代码中就有3个不同的任务：获取网页内容、查找指定网页内容、发送邮件。要保证一个函数只做一件事，就要尽量保证抽象层级的一致性，所有的语句尽量在一个粒度上。如上例既有 `http.getfile()` 这样较高层级抽象的语句，也有细粒度的字符处理语句。同时在一个函数中处理多件事情也不利于代码的重用，在上例中，如果程序中还有类似发送邮件的需求，必然造成代码的冗余。

最后，根据以上几点原则，上面对本节最开始处的代码进行修改，来看看修改后的代码是不是可读性要好一些。

```
def GetContent(ServerAdr, PagePath):
    http = httpplib.HTTP(ServerAdr)
    http.putrequest('GET', PagePath)
    http.putheader('Accept', 'text/html')
    http.putheader('Accept', 'text/plain')
    http.endheaders()
    httpcode, httpmsg, headers = http.getreply()
    if httpcode != 200:
        raise "Could not get document: Check URL and Path."
    doc = http.getfile()
    data = doc.read()                # read file
    doc.close()
    return data
def ExtractData(inputstring, start_line, end_line):
    lstr=inputstring.splitlines()    #split
    j=0                               #set counter to
    zero
    for i in lstr:
        j=j+1
        if i.strip() == start_line: slice_start=j        #find slice start
        elif i.strip() == end_line: slice_end=j          #find slice end
    return lstr[slice_start:slice_end]                    #return slice
def SendEmail(sender, receiver, smtpserver, username, password, content):
    subject = "Contented get from the web"
    msg = MIMEText(content, 'plain', 'utf-8')
    msg['Subject'] = Header(subject, 'utf-8')
    smtp = smtplib.SMTP()
```

```
smtp.connect(smtpserver)
smtp.login(username, password)
smtp.sendmail(sender, receiver, msg.as_string())
smtp.quit()
```

Python中函数设计的好习惯还包括：不要在函数中定义可变对象作为默认值，使用异常替换返回错误，保证通过单元测试等。

建议7：将常量集中到一个文件

Python中存在常量吗？相信很多人的答案是否定的。实际上Python的内建命名空间是支持一小部分常量的，如我们熟悉的True、False、None等，只是Python没有提供定义常量的直接方式而已。那么，在Python中应该如何使用常量呢？一般来说有以下两种方式：

- 通过命名风格来提醒使用者该变量代表的意义为常量，如常量名所有字母大写，用下划线连接各个单词，如MAX_OVERFLOW、TOTAL。然而这种方式并没有实现真正的常量，其对应的值仍然可以改变，这只是一种约定俗成的风格。

- 通过自定义的类实现常量功能。这要求符合“命名全部为大写”和“值一旦绑定便不可再修改”这两个条件。下面是一种较为常见的解决方法，它通过对常量对应的值进行修改时或者命名不符合规范时抛出异常来满足以上常量的两个条件。

```
class _const:
    class ConstError(TypeError): pass
    class ConstCaseError(ConstError): pass
    def __setattr__(self, name, value):
        if self.__dict__.has_key(name):
            raise self.ConstError, "Can't change const.%s" % name
        if not name.isupper():
            raise self.ConstCaseError, \
                'const name "%s" is not all uppercase' % name
        self.__dict__[name] = value
import sys
sys.modules[__name__]=_const()
```

如果上面的代码对应的模块名为const，使用的时候只需要import const，便可以直接定义常量了，如以下代码：

```
import const
const.COMPANY = "IBM"
```

上面的代码中常量一旦赋值便不可再更改，因此`const.COMPANY = "SAP"`会抛出`const.ConstError`异常，而常量名称如果小写，如`const.name = "Python"`，也会抛出`const.ConstCaseError`异常。

无论采用哪一种方式来实现常量，都提倡将常量集中到一个文件中，因为这样有利于维护，一旦需要修改常量的值，可以集中统一进行而不是逐个文件去检查。采用第二种方式实现的常量可以这么做：将存放常量的文件命名为`constant.py`，并在其中定义一系列常量。

```
class _const:
    class ConstError(TypeError): pass
    class ConstCaseError(ConstError): pass
    def __setattr__(self, name, value):
        if self.__dict__.has_key(name):
            raise self.ConstError, "Can't change const.%s" % name
        if not name.isupper():
            raise self.ConstCaseError, \
                'const name "%s" is not all uppercase' % name
        self.__dict__[name] = value
import sys
sys.modules[__name__]=_const()
import const
const.MY_CONSTANT = 1
const.MY_SECOND_CONSTANT = 2
const.MY_THIRD_CONSTANT = 'a'
const.MY_FORTH_CONSTANT = 'b'
```

当在其他模块中引用这些常量时，按照如下方式进行即可：

```
from constant import const
print const.MY_SECOND_CONSTANT
print const.MY_THIRD_CONSTANT*2
print const.MY_FORTH_CONSTANT+'5'
```

第2章 编程惯用法

编程语言通常都有其惯用法，掌握这些惯用法能够帮助我们写出更为专业和精简的程序，所以说，掌握这些惯用法是非常必要的。本章主要讨论Python中一些常见的编程惯用法以及其所涉及的一些编程思想。

建议8： 利用assert语句来发现问题

断言（**assert**）在很多语言中都存在，它主要为调试程序服务，能够快速方便地检查程序的异常或者发现不恰当的输入等，可防止意想不到的情况出现。Python自1.5版本开始引入断言语句，其基本语法如下：

```
assert expression1 [," expression2"]
```

其中计算**expression 1**的值会返回**True**或者**False**，当值为**False**的时候会引发**AssertionError**，而**expression 2**是可选的，常用来传递具体的异常信息。

来看一个简单的使用例子：

```
>>> x =1
>>> y =2
>>> assert x == y,"not equals"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: not equals
>>>
```

在执行过程中它实际相当于如下代码：

```
>>> x =1
>>> y =2
>>> if __debug__ and not x == y:
...     raise AssertionError("not equals")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AssertionError: not equals
>>>
```

对Python中使用断言需要说明如下：

1) `__debug__`的值默认设置为`True`，且是只读的，在`Python2.7`中还无法修改该值。

2) 断言是有代价的，它会对性能产生一定的影响，对于编译型的语言，如`C/C++`，这也许并不那么重要，因为断言只在调试模式下启用。但`Python`并没有严格定义调试和发布模式之间的区别，通常禁用断言的方法是在运行脚本的时候加上`-O`标志，这种方式带来的影响是它并不优化字节码，而是忽略与断言相关的语句。如：

```
def foo(x):  
    assert x  
foo(0)
```

运行`python asserttest.py`如下：

```
Traceback (most recent call last):  
  File "asserttest.py", line 4, in <module>  
    foo(0)  
  File "asserttest.py", line 2, in foo  
    assert x  
AssertionError
```

加上`-O`的参数：`python -O asserttest.py`便可以禁用断言。

断言实际是被设计用来捕获用户所定义的约束的，而不是用来捕获程序本身错误的，因此使用断言需要注意以下几点：

1) 不要滥用，这是使用断言最基本的原则。若由于断言引发了异常，通常代表程序中存在`bug`。因此断言应该使用在正常逻辑不可到达的地方或正常情况下总是为真的场合。

2) 如果`Python`本身的异常能够处理就不要再使用断言。如对于类似于数组越界、类型不匹配、除数为0之类的错误，不建议使用断言来

进行处理。下面的例子中使用断言就显得多余，因为如果传入的参数一个为字符串，另一个为数字或者列表，本身就会抛出**TypeError**。

```
>>> def stradd(x,y):  
...     assert isinstance(x,basestring)  
...     assert isinstance(y,basestring)  
...     return x+y
```

3) 不要使用断言来检查用户的输入。如对于一个数字类型，如果根据用户的设计该值的范围应该是2~10，较好的做法是使用条件判断，并在不符合条件的时候输出错误提示信息。

4) 在函数调用后，当需要确认返回值是否合理时可以使用断言。

5) 当条件是业务逻辑继续下去的先决条件时可以使用断言。如list1和其副本list2，业务继续下去的条件是这两个list必须是一样的，但由于某些不可控因素，如使用了浅拷贝而list1中含有可变对象等，就可以使用断言来判断这两者的关系，如果不相等，则继续运行后面的程序意义不大。

建议9：数据交换值的时候不推荐使用中间变量

交换两个变量的值，大家熟悉的代码如下：

```
>>> temp = x
>>> x = y
>>> y = temp
```

实际上，在Python中还有更简单、更Pythonic的实现方式，代码如下：

```
>>> x,y = y,x
```

上面的实现方式不需要借助任何中间变量并且能够获取更好的性能。我们来简单测试一下。

```
>>> from timeit import Timer
>>> Timer('temp = x;x = y;y = temp','x=2;y =3').timeit()
0.10689428530212189
>>> Timer('x,y = y,x','x=2;y=3').timeit()
0.08492583713832502
```

从测试结果可以看出，第二种方式耗费的时间更少，并且由于不需要借助中间变量，代码更为简洁，是值得推荐的一种方式。那么，为什么第二种方式可以做到更优呢？这要从Python表达式计算的顺序说起。一般情况下Python表达式的计算顺序是从左到右，但遇到表达式赋值的时候表达式右边的操作数先于左边的操作数计算，因此表达式`expr3`，`expr4 = expr1, expr2`的计算顺序是`expr1, expr2 → expr3, expr4`。因此对于表达式`x, y=y, x`，其在内存中执行的顺序如下：

1) 先计算右边的表达式 y, x ，因此先在内存中创建元组 (y, x) ，其标示符和值分别为 y, x 及其对应的值，其中 y 和 x 是在初始化时已经存在于内存中的对象。

2) 计算表达式左边的值并进行赋值，元组被依次分配给左边的标示符，通过解压缩（**unpacking**），元组第一标识符（为 y ）分配给左边第一个元素（此时为 x ），元组第二个标识符（为 x ）分配给第二个元素（此时为 y ），从而达到 x, y 值交换的目的。

更深入一点我们从Python生成的字节码来分析。Python的字节码是一种类似汇编指令的中间语言，但是一个字节码指令并不是对应一个机器指令。我们通过以下dis模块的来进行分析：

```
>>> import dis
>>> def swap1():
...     x = 2
...     y = 3
...     x, y = y, x
...
>>> def swap2():
...     x = 2
...     y = 3
...     temp = x
...     x = y
...     y = temp
...
>>> print 'swap1():'
swap1():
>>> dis.dis(swap1)
 2           0 LOAD_CONST           1 (2)
              3 STORE_FAST          0 (x)
 3           6 LOAD_CONST           2 (3)
              9 STORE_FAST          1 (y)
 4          12 LOAD_FAST            1 (y)
              15 LOAD_FAST            0 (x)
              18 ROT_TWO
              19 STORE_FAST          0 (x)
              22 STORE_FAST          1 (y)
              25 LOAD_CONST           0 (None)
              28 RETURN_VALUE

>>> print 'swap2():'
swap2():
>>> dis.dis(swap2)
 2           0 LOAD_CONST           1 (2)
              3 STORE_FAST          0 (x)
 3           6 LOAD_CONST           2 (3)
              9 STORE_FAST          1 (y)
 4          12 LOAD_FAST            0 (x)
              15 STORE_FAST          2 (temp)
 5          18 LOAD_FAST            1 (y)
```

6	21	STORE_FAST	0	(x)
	24	LOAD_FAST	2	(temp)
	27	STORE_FAST	1	(y)
	30	LOAD_CONST	0	(None)
	33	RETURN_VALUE		

通过字节码可以看出，区别主要集中在swap1函数的第4行和swap2函数的第4~6行代码，其中swap1的第4行代码对应的字节码中有2个LOAD_FAST指令、2个STORE_FAST指令和1个ROT_TWO指令，而swap2函数对应的第4~6行代码中共生成了3个LOAD_FAST指令和3个STORE_FAST指令。而指令ROT_TWO的主要作用是交换两个栈的最顶层元素，它比执行一个LOAD_FAST+STORE_FAST指令更快。

建议10：充分利用Lazy evaluation的特性

Lazy evaluation常被译为“延迟计算”或“惰性计算”，指的是仅仅在真正需要执行的时候才计算表达式的值。充分利用Lazy evaluation的特性带来的好处主要体现在以下两个方面：

1) 避免不必要的计算，带来性能上的提升。对于Python中的条件表达式if x and y，在x为false的情况下y表达式的值将不再计算。而对于if x or y，当x的值为true的时候将直接返回，不再计算y的值。因此编程中应该充分利用该特性。下面的例子用于判断一个单词是不是指定的缩写形式。

```
from time import time
t = time()
abbreviations = ['cf.', 'e.g.', 'ex.', 'etc.', 'fig.', 'i.e.', 'Mr.', 'vs.']
for i in xrange(1000000):
    for w in ('Mr.', 'Hat', 'is', 'chasing', 'the', 'black', 'cat', '.'):
        if w in abbreviations:
            #if w[-1] == '.' and w in abbreviations:
                pass
print "total run time:"
print time()-t
```

如果使用注释行代替第一个if，运行的时间大约会节省10%。因此在编程过程中，如果对于or条件表达式应该将值为真可能性较高的变量写在or的前面，而and则应该推后。

2) 节省空间，使得无限循环的数据结构成为可能。Python中最典型的使用延迟计算的例子就是生成器表达式了，它仅在每次需要计算的时候才通过yield产生所需要的元素。斐波那契数列在Python中实现起来就显得相当简单，而while True也不会导致其他语言中所遇到的无限循环的问题。

```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
>>> from itertools import islice
>>> print list(islice(fib(), 5))
[0, 1, 1, 2, 3]
```

Lazy evaluation并不是一个很大、很新鲜的话题，但古人云“不积跬步无以至千里”，小小的改进便能写出更为优化的代码，何乐而不为呢？

建议11：理解枚举替代实现的缺陷

关于枚举最经典的例子大概非季节和星期莫属了，它能够以更接近自然语言的方式来表达数据，使得程序的可读性和可维护性大大提高。然而，很不幸，也许你习惯了其他语言中的枚举类型，但在Python3.4以前却并不提供。关于要不要加入枚举类型的问题就引起了不少讨论，在PEP354中曾提出增加枚举的建议，但被拒绝。于是人们充分利用Python的动态性这个特征，想出了枚举的各种替代实现方式。

1) 使用类属性。

```
>>> class Seasons:
...     Spring = 0
...     Summer = 1
...     Autumn = 2
...     Winter = 3
...
>>> print Seasons.Spring
```

上面的例子可以直接简化为：

```
>>> class Seasons:
...     Spring, Summer, Autumn, Winter = range(4)
...

```

2) 借助函数。

```
>>> def enum(*posarg, **keysarg):
...     return type("Enum", (object,), dict(zip(posarg, xrange(len(posarg))), **
keysarg))
...
>>> Seasons = enum("Spring", "Summer", "Autumn", Winter=1)
>>> Seasons.Spring
0
```

3) 使用collections.namedtuple。

```
>>> Seasons = namedtuple('Seasons', 'Spring Summer Autumn Winter')._make(range(4))
>>> print Seasons.Spring
0
```

Python中枚举的替代实现方式远不止上述这些，在此就不一一列举了。那么，既然枚举在Python中有替代的实现方式，为什么人们还要执着地提出各自建议要求语言实现枚举呢？显然，这些替代实现有其不合理的地方。

·允许枚举值重复。我们以collections.namedtuple为例，下面的例子中枚举值Spring与Autumn相等，但却不会提示任何错误。

```
>>> Seasons._replace(Spring =2)
Seasons(Spring=2, Summer=1, Autumn=2, Winter=3) #Spring
和Autumn
的值相等，都为2
```

·支持无意义的操作。

```
>>> Seasons.Summer+Seasons.Autumn == Seasons.Winter
True #Seasons.Summer+Seasons.Autumn
相加无任何实际意义
```

实际上Python2.7以后的版本还有另外一种替代选择——使用第三方模块fluhl.enum，它包含两种枚举类：一种是Enum，只要保证枚举值唯一即可，对值的类型没限制；还有一种是IntEnum，其枚举值为int型。

```
>>> from fluhl.enum import Enum
>>> class Seasons(Enum):                                     #
继承自Enum
定义枚举
...     Spring = "Spring"
...     Summer = 2
...     Autumn = 3
```

```
...     Winter = 4
...
>>> Seasons = Enum('Seasons', 'Spring Summer Autumn Winter')
```

`fluf.enum`提供了`__members__`属性，可以对枚举名称进行迭代。

```
>>> for member in Seasons.__members__:
...     print member
...
Spring
Summer
Autumn
Winter
```

可以直接使用`value`属性获取枚举元素的值，如：

```
>>> print Seasons.Summer.value
2
```

`fluf.enum`不支持枚举元素的比较。

```
>>> Seasons.Summer < Seasons.Autumn      #fluf.enum
不支持无意义的操作
Traceback (most recent call last):
... ..
...     raise NotImplementedError
NotImplementedError
```

更多关于`fluf.enum`的使用可以参考网页
<http://Pythonhosted.org/fluf.enum/docs/using.html>的内容。

值得一提的是，Python3.4中根据PEP435的建议终于加入了枚举`Enum`，其实现主要参考实现`fluf.enum`，但两者之间还是存在一些差别，如`fluf.enum`允许枚举继承，而`Enum`仅在父类没有任何枚举成员的时候才允许继承等，读者可以仔细阅读PEP435了解更多详情。另外，如果要在Python3.4之前的版本中使用枚举`Enum`，可以安装`Enum`

的向后兼容包enum34，下载地址为
<https://pypi.python.org/pypi/enum34>。

建议12：不推荐使用type来进行类型检查

作为动态性的强类型脚本语言，Python中的变量在定义的时候并不会指明具体类型，Python解释器会在运行时自动进行类型检查并根据需要进行隐式类型转换。按照Python的理念，为了充分利用其动态性的特征是不推荐进行类型检查的。如下面的函数add()，在无需对参数进行任何约束的情况下便可以轻松地实现字符串的连接、数字的加法、列表的合并等多种功能，甚至处理复数都非常灵活。解释器能够根据变量类型的不同调用合适的内部方法进行处理，而当a、b类型不同而两者之间又不能进行隐式类型转换时便抛出TypeError异常。

```
def add(a, b):
    return a+b
print add(1,2j)
①复数相加
print add('a','b')
②字符串连接
print add(1,2)
③整数
print add(1.0,2.3)
④浮点数处理
print add([1,2],[2,3])
⑤处理列表
print add(1,'a')
⑥不同类型
```

输出如下：

```
(1+2j)
ab
3
3.3
[1, 2, 2, 3]
Traceback (most recent call last):
  File "tests.py", line 9, in <module>
    print add(1,'a')
  File "tests.py", line 2, in add
    return a+b
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

不刻意进行类型检查，而是在出错的情况下通过抛出异常来进行处理，这是较为常见的方式。但实际应用中为了提高程序的健壮性，仍然会面临需要进行类型检查的情景。那么使用什么方法呢？很容易想到，使用`type()`。

内建函数`type(object)`用于返回当前对象的类型，如`type(1)`返回`<type 'int'>`。因此可以通过与Python自带模块`types`中所定义的名称进行比较，根据其返回值确定变量类型是否符合要求。例如判断一个变量`a`是不是`list`类型可以使用以下代码：

```
if type(a) is types.ListType:
```

所有基本类型对应的名称都可以在`types`模块中找到，如`types.BooleanType`、`types.IntType`、`types.StringType`、`types.DictType`等。然而使用`type()`函数并不是就意味着可以高枕无忧了，主张“不推荐使用`type`来进行变量类型检查”是有一定的缘由的。来看几个例子。

示例一：下例中用户类`UserInt`继承`int`类实现定制化，它不支持操作符`+=`。具体代码如下：

```
import types
class UserInt(int) :                #int
    为用户Int
    的基类
    def __init__(self, val=0) :
        self._val = int(val)
    def __add__(self, val) :        #
    实现整数的加法
        if isinstance(val, UserInt):
            return UserInt(self._val + val._val)
        return self._val + val
    def __iadd__(self, val) :
    ①UserInt
    不支持+=
    操作
        raise NotImplementedError("not support operation")
    def __str__(self) :
        return str(self._val)
    def __repr__(self) :
        return 'Integer(%s)' %self._val
n = UserInt()
```

```
print n
m = UserInt(2)
print m
print n+m
print type(n) is types.IntType
②使用type
进行类型判断
```

程序输出如下:

```
0
2
2
False
```

上例标注②处输出**False**，这说明`type()`函数认为**n**并不是**int**类型，但**UserInt**继承自**int**，显然这种判断不合理。由此可见[基于内建类型扩展的用户自定义类型](#)，`type`函数并不能准确返回结果。

示例二：在古典类中，所有类的实例的**type**值都相等。

```
>>> class A:
...     pass
...
>>> a = A()
>>> class B:
...     pass
...
>>> b = B()
>>> type(a) == type(b)                                #
判断两者类型是否相等
True
```

在古典类中，任意类的实例的**type()**返回结果都是<type 'instance'>。这种情况下使用**type()**函数来确定两个变量类型是否相同显然结果会与我们所理解的大相径庭。

因此对于内建的基本类型来说，也许使用**type()**进行类型检查问题不大，但在某些特殊场合**type()**方法并不可靠。那么究竟应怎样来约束用户的输入类型从而使之与我们期望的类型一致呢？答案是：如果类

型有对应的工厂函数，可以使用工厂函数对类型做相应转换，如 `list(listing)`、`str(name)`等，否则可以使用`isinstance()`函数来检测，其原型如下：

```
isinstance(object, classinfo)
;
```

其中，`classinfo`可以为直接或间接类名、基本类型名称或者由它们组成的元组，该函数在`classinfo`参数错误的情况下会抛出`TypeError`异常。

`isinstance`基本用法举例如下：

```
>>> isinstance(2, float)
False
>>> isinstance("a", (str, unicode))
True
>>> isinstance((2, 3), (str, list, tuple))
True
①支持多种类型列表
```

因此示例一中可以将`print type(n) is types.IntType`改为`print isinstance(n, int)`，以获取正确的结果。

建议13：尽量转换为浮点类型后再做除法

GPA（平均成绩绩点）在出国留学或者奖学金申请中都占有重要的地位。**GPA**算法有多种形式，其中标准计算方法是将大学成绩乘以课程学分并求和再乘以4，再除以总学分与100之积，一般精确到小数点后两位。假如学生A的各门课程成绩如下：

A课程4学分，成绩96（等级A，绩点4）；B课程3学分，成绩85（等级B，绩点3）

C课程5学分，成绩98（等级A，绩点4）；D课程2学分，成绩70（等级C，绩点2）

那么该学生的**GPA**是多少呢？很容易的算术问题对吧，小学生都会！那么你算出来的结果是多少？Python计算出的结果又是多少呢？

```
>>> gpa = ((4*96+3*85+5*98+2*70)*4)/((4+3+5+2)*100)
>>> print gpa
3
```

上面的结果跟你计算出的答案一致吗？显然是否定的，你的计算结果为3.62571428571，即使四舍五入保留小数点后两位也应该是3.63。如果有所大学规定的最低**GPA**是3.5的话，用Python作为**GPA**计算工具的话可就实实在在会误人前程了。问题出现在哪里呢？这要回到Python设计之初。

Python在最初的设计过程中借鉴了C语言的一些规则，比如选择C的long类型作为Python的整数类型，double作为浮点类型等。同时标准的算术运算，包括除法，返回值总是和操作数类型相同。作为静态类

型语言，C语言中这一规则问题不大，因为变量都会预先申明类型，当类型不符的时候，编译器也会尽可能进行强制类型转换，否则编译会报错。但Python作为一门高级动态语言并没有类型申明这一说，因此在上面的例子中你不能提前申明返回的计算结果为浮点数，当除法运算中两个操作数都为整数的时候，其返回值也为整数，运算结果将直接截断，从而在实际应用中造成潜在的质的误差。

Python中除了除法运算之外，整数和浮点数的其他操作行为还是一致的，因此这容易让人产生一种误解，数值的计算与具体操作数的类型（整数还是浮点数）无关，但事实上对于整数除法这是编程过程中潜在的一个危险，因为当你编写一个函数时，即使你希望调用者传入的是浮点类型，但如果不在函数入口进行类型检查或者转换，就无法阻止函数调用者传递整数参数，而往往这种类型的错误还不容易发觉。因此推荐的做法之一是[当涉及除法运算的时候尽量先将操作数转换为浮点类型再做运算](#)。

```
>>> gpa = float(((4*96+3*85+5*98+2*70)*4))/float(((4+3+5+2)*100))
>>> print gpa
3.62571428571
```

当然随着Python语言的发展，对整数除法问题也做了一定的修正，在Python3中这个问题已经不存在了。Python3之前的版本可以通过[from __future__ import division](#)机制使整数除法不再截断，这样即使不进行浮点类型转换，输出结果也是正确的（请读者自行试验）。

最后，还需要说明一点，上例中是使用浮点数才精确，但下列场景又变成了浮点数可能是不准确的。先来看以下代码：

```
>>> i=1
>>> while i!=1.5:
...     i = i +0.1
...     print i
```

上面的代码输出会是多少？正确的答案是这段代码会导致无限循环。为什么呢？因为在计算机的世界里，浮点数的存储规则决定了不是所有的浮点数都能准确表示，有些是不准确的，只是无限接近。如0.1转换为二进制表示形式则为0.000110011001.....后面1001无限循环。在内存中根据浮点数位数规定，多余部分直接截断，因此当循环到第5次的时候i的实际值为1.5000000000000004（读者可以逐步调试进行验证），则条件表达式*i* != 1.5显然为True，循环陷入无终止状态。对于浮点数的处理，要记住其运算结果可能并不是完全准确的。如果计算对精度要求较高，可以使用Decimal来进行处理或者将浮点数尽量扩大为整数，计算完毕之后再转换回去。而对于在while中使用*i* != 1.5这种条件表达式更是要避免的，浮点数的比较同样最好能够指明精度。

建议14：警惕eval()的安全漏洞

如果你了解JavaScript或者PHP等，那么你一定对eval()有所了解。如果你并没有接触过也没关系，eval()函数的使用非常简单。

```
>>> eval("1+1==2")           #
进行判断
True
>>>
>>> eval("'A'+'B'")          #
字符连接
'AB'
>>> eval("1+2")              #
数字相加
3
>>>
```

Python中eval()函数将字符串str当成有效的表达式来求值并返回计算结果。其函数声明如下：

```
eval(expression[, globals[, locals]])
```

其中，参数globals为字典形式，locals为任何映射对象，它们分别表示全局和局部命名空间。如果传入globals参数的字典中缺少__builtins__的时候，当前的全局命名空间将作为globals参数输入并且在表达式计算之前被解析。locals参数默认与globals相同，如果两者都省略的话，表达式将在eval()调用的环境中执行。

“eval is evil”（eval是邪恶的），这是一句广为人知的对eval的评价，它主要针对的是eval()的安全性。那么eval存在什么样的安全漏洞呢？来看一个简单的例子：

```
import sys
from math import *
def ExpCalcBot(string):
```

```

    try:
        print 'Your answer is',eval(user_func)          #
计算输入的值
    except NameError:
        print "The expression you enter is not valid"
print 'Hi,I am ExpCalcBot. please input your experssion or enter e to end'
inputstr = ''
while 1:
    print 'Please enter a number or operation. Enter c to complete. : '
    inputstr = raw_input()
    if inputstr == str('e') :                            #
遇到输入为e
的时候退出
        sys.exit()
    elif repr(inputstr) != repr(''):
        ExpCalcBot(inputstr)
        inputstr = ''

```

上面这段代码的主要功能是：根据用户的输入，计算Python表达式的值。它有什么问题呢？如果用户都是素质良好，没有不良目的的话，那么这段程序也许可以满足基本需求。比如，输入`1+sin(20)`会输出结果`1.91294525073`。但如果它是一个Web页面的后台调用（当然，你需要做一定的修改），由于网络环境下运行它的用户并非都是可信的，问题就出现了。因为`eval()`可以将任何字符串当做表达式求值，这也就意味着有空子可钻。上面的例子中假设用户输入`__import__("os").system("dir")`，会有什么样的输出呢？你会惊讶地发现它会显示当前目录下的所有文件列表，输出如下：

```

C:\test>python tests.py
Hi,I am ExpCalcBot. please input your experssion or enter e to end
Please enter a number or operation. Enter c to complete. :
__import__("os").system("dir")
Your answer is
驱动器 C
中的卷是 SYSTEM
卷的序列号是 A0E0-1A46
C:\test
的目录
2013/07/31 12:28 <DIR> .
2013/07/31 12:28 <DIR> ..
2012/07/24 08:11 859,919 2012-07-24-010.jpg
2012/07/24 08:11 1,037,015 2012-07-24-011.jpg
2012/07/24 08:19 1,242,667 2012-07-24-012.jpg
2013/07/31 12:26 0 test.txt
2013/07/31 12:27 503 tests.py
5
个文件 3,140,104
字节
2
个目录 422,254,702,592
可用字节

```

```
0
Please enter a number or operation. Enter c to complete. :
```

于是顿时，有人的“坏心眼”来了，他输入了如下字符串，可悲的事情发生了，当前目录下的所有文件都被删除了，包括test.py，而这一切没有任何提示，悄无声息。

```
__import__("os").system("del * /Q")
!!! 不要轻易在你的计算机上尝试
Your answer is 0
```

试想，在网络环境下这是不是很危险？也许你会辩护，那是因为你没有在globals参数中禁止全局命名空间的访问。好，我们按照你说的来试验一下：将函数ExpCalcBot修改一下，其中math_fun_list限定为几个常用的数学函数。修改后的函数如下：

```
def ExpCalcBot(string):
    try:
        math_fun_list = ['acos', 'asin', 'atan', 'cos', 'e', 'log', 'log10', 'pi',
                        'pow', 'sin', 'sqrt', 'tan']
        math_fun_dict = dict([ (k, globals().get(k)) for k in math_fun_list ])
        #
        形成可以访问的函数的字典
        print 'Your answer is',eval(string,{"__builtins__": None},math_fun_dict)
    except NameError:
        print "The expression you enter is not valid"
```

再次运行程序（请读者自行试验）你会惊喜地发现上面的命令被看着无效表达式，你的辩护是对的，这确实是我们想要的。很好，安全问题不再是个问题！但仔细想想真是这样的吗？试试输入以下字符：

```
[c for c in ().__class__.__bases__[0].__subclasses__() if c.__name__ == 'Quitter']
[0]()[0]()
```

().__class__.__bases__[0].__subclasses__()用来显示object类的所有子类。类Quitter与"quit"功能绑定，因此上面的输入会直接导致程序退

出。

注：你可以在Python的安装目录下的Lib\site.py中找到其类的定义。读者也可以自行在Python解释器中输入
`print().__class__.__bases__[0].__subclasses__()`看看输出结果是什么。

因此对于有经验的侵入者来说，他可能会有一系列强大的手段，使得eval可以解释和调用这些方法，从而带来更大的破坏。此外，eval()函数也给程序的调试带来一定困难，要查看eval()里面表达式具体的执行过程很难。因此在实际应用过程中如果使用对象不是信任源，应该尽量避免使用eval，在需要使用eval的地方可用安全性更好的ast.literal_eval替代。literal_eval函数具体详情可以参考文档http://docs.python.org/2/library/ast.html#ast.literal_eval。上面的例子请读者使用literal_eval自行试验，你会体会得更加深刻。

建议15：使用enumerate()获取序列迭代的索引和值

基本上所有的项目中都存在对序列进行迭代并获取序列中的元素进行处理的场景。这是一个非常普通而且简单的需求，相信很多人一口气能写出N种实现方法。举例如下。

方法一 在每次循环中对索引变量进行自增。

```
li = ['a', 'b', 'c', 'd', 'e']
index=0
for i in li:
    print "index:",index,"element:",i
    index+=1
```

方法二 使用range()和len()方法结合。

```
li = ['a', 'b', 'c', 'd', 'e']
for i in range(len(li)):
    print "index:",i,"element:",li[i]
```

方法三 使用while循环，用len()获取循环次数。

```
li = ['a', 'b', 'c', 'd', 'e']
index=0
while index < len(li):
    print "index:",index,"element:",li[index]
    index+=1
```

方法四 使用zip()方法。

```
li = ['a', 'b', 'c', 'd', 'e']
for i, e in zip(range(len(li)), li):
    print "index:",i,"element:",e
```

方法五 使用enumerate()获取序列迭代的索引和值。

```
li = ['a', 'b', 'c', 'd', 'e']
for i,e in enumerate(li):
    print "index:",i,"element:",e
```

这里推荐的是使用方法五，因为它代码清晰简洁，可读性最好。函数enumerate()是在Python2.3中引入的，主要是为了解决在循环中获取索引以及对应值的问题。它具有一定的惰性（lazy），每次仅在需要的时候才会产生一个(index,item)对。其函数签名如下：

```
enumerate(sequence, start=0)
```

其中，sequence可以为序列，如list、set等，也可以为一个iterato或者任何可以迭代的对象，默认的start为0，函数返回本质上为一个迭代器，可以使用next()方法获取下一个迭代元素，如下所示：

```
>>> li = ['a', 'b', 'c', 'd', 'e']
>>> print enumerate(li)
<enumerate object at 0x00373E18>
>>> e = enumerate(li)
>>> e.next()
(0, 'a')
>>> e.next()
(1, 'b')
```

enumerate()函数的内部实现非常简单，enumerate(sequence,start=0)实际相当于如下代码：

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

因此利用这个特性用户还可以实现自己的`enumerate()`函数。比如，`myenumerate()`以反序的方式获取序列的索引和值。

```
def myenumerate(sequence):
    n = -1
    for elem in reversed(sequence):
        yield len(sequence)+n, elem
        n = n-1
li = ['a', 'b', 'c', 'd', 'e']
for i,e in myenumerate(li):
    print "index:",i,"element:",e
```

程序输出如下:

```
index: 4 element: e
index: 3 element: d
index: 2 element: c
index: 1 element: b
index: 0 element: a
```

需要提醒的是，对于字典的迭代循环，`enumerate()`函数并不适合，虽然在使用上并不会提示错误，但输出的结果与期望的大相径庭，这是因为字典默认被转换成了序列进行处理。

```
personinfo = {'name': 'Jon', 'age': '20', 'hobby': 'football'}
for k, v in enumerate(personinfo):
    print k,v
输出为:
0 hobby
1 age
2 name
```

要获取迭代过程中字典的`key`和`value`，应该使用如下`iteritems()`方法:

```
for k,v in personinfo.iteritems():
    print k,":",v
```

建议16：分清==与is的适用场景

在判断两个字符串是否相等的时候，混用is和==是很多初学者经常犯的错误，造成的结果是程序在不同情况下表现不一。先来看一个例子：

```
>>> a = "Hi"
>>> b = "Hi"
>>> a is b
True
>>> a == b      #is
和 ==
结果一样
True
>>> a1 = "I am using long string for testing"
>>> b1 = "I am using long string for testing"
>>> a1 is b1     #is
的结果为False
False
>>> a1 == b1     # ==
的结果为True
。两者并不一样
True
>>> str1 = "string"
>>> str2 = "".join(['s','t','r','i','n','g'])
>>> print str2
string
>>> str1 is str2
False
>>> str1 == str2      #==
和is
的结果在这种情况下也不一样
True
```

造成这种奇怪现象的原因是什么呢？为什么在有些情况下is和==输出相同而在有些情况下又不相同呢？我们分析一下：首先通过id()函数来看看这些变量在内存中具体的存储空间，为了方便讨论问题，用表2-1来表示上例具体结果。

表2-1 不同变量组id()以及is和==的求值结果

	id()	is	==
a = "Hi" ①	14085224	True	True
b = "Hi" ①	14085224		
a1 = "I am using long string for testing" ②	13003368	False	True
b1 = "I am using long string for testing" ②	14078152		
str1 = "string" ③	13256448	False	True
str2 = "".join(['s','t','r','i','n','g']) ③	14056544		

从表格中可以清晰地看到，is和==在验证两个字符串是否相等的时候表现确实不一致，显然混用或者将它们等同起来是存在风险的。那么字符串的比较到底是用is还是用==呢？先来看看Python官方文档中对这两种操作的如下表2-2所示。

表2-2 两种操作的意义

操 作 符	意 义
is	object identity
==	equal

is表示的是对象标示符（object identity），而==表示的意思是相等（equal），显然两者不是一个东西。实际上，造成上面输出结果不一致的根本原因在于：is的作用是用来检查对象的标示符是否一致的，也就是比较两个对象在内存中是否拥有同一块内存空间，它并不适合用来判断两个字符串是否相等。x is y仅当x和y是同一个对象的时候才返回True，x is y 基本相当于id(x) == id(y)。而==才是用来检验两个对象的值是否相等的，它实际调用内部__eq__()方法，因此a == b相当于a.__eq__(b)，所以==操作符是可以被重载的，而is不能被重载。一般情况下，如果x is y为True的话x == y的值也为True（特殊情况除外，如NaN，a = float('NaN'); a is a 为True，a==a为false），反之则不然。

弄清楚了is和==之间的区别，再来看上述表格中的输出也就不难理解了。但如果再细心一点也许会发现第1组（标注①）中a和b的id值一样，也就是说它们在内存中是同一个对象，而第二组（标注②）中a1和

b1的id值却不一样。这又是为什么呢？这是Python中的string interning（字符串驻留）机制所决定的：对于较小的字符串，为了提高系统性能会保留其值的一个副本，当创建新的字符串的时候直接指向该副本即可。因此标注①中“Hi”在系统内存中实际上只有一个副本，所以a和b的id值是一样的；而标注②中a1和b1是长字符串，并不会驻留，Python内存中各自创建了对象来表示a1和b1，这两个对象拥有相同的内容但对象标示符却不相同，所以==的值为True而is的值为False。



注意

判断两个对象相等应该使用==而不是is。

建议17：考虑兼容性，尽可能使用Unicode

Python内建的字符串有两种类型：`str`和`Unicode`，它们拥有共同的祖先`basestring`。其中`Unicode`是Python2.0中引入的一种新的数据类型，所有的`Unicode`字符串都是`Unicode`类型的实例。创建一个`Unicode`字符相对简单。

```
>>> strUnicode = u"unicode
字符串" #
前面加u
表示Unicode
>>> strUnicode
u'unicode \u5b57\u7b26\u4e32'
>>> print strUnicode
unicode
字符串
>>> type(strUnicode)
<type 'unicode'>
>>> type(strUnicode).__bases__
(<type 'basestring'>,)

```

Python中为什么需要加入对`Unicode`的支持呢？我们先来了解一下`Unicode`相关的背景知识。

在`Unicode`之前，最早的`ASCII`编码用一个字节（8bit，最高位为0）只能表示128个字符，如英文大小写字符、数字以及其他符号等。但世界上显然不只有一种语言，不同种语言所包含的字符数量也不相同，对于很多语言来说128个字符数是远远不够的，即使对`ASCII`进行扩展，256个字符也不能满足要求。于是出现了各种不同的字符编码系统，如我国表示汉字编码的`GBK`。但这又引入了一个新的问题：不同编码系统之间存在冲突。在两种不同的编码系统中，相同的编码可能代表不同的意义或者不同的编码代表相同的字符，从而导致不同平台、不同语言之间的文本无法很好地进行转换。比如，“我”字在`GB2312`中表示为`0x4650`，而繁体中文`Big5`中的编码为`0XA7DA`，而

0XA7DA在GB2312中却表示“賧”，乱码由此产生。要解决这个问题，必须为不同的文字分配统一编码，Unicode（Universal Multiple-Octet Coded Character Set）由此产生，它也被称作万国码，Unicode为每种语言设置了唯一的二进制编码表示方式，提供从数字代码到不同语言字符集之间的映射，从而可以满足跨平台、跨语言之间的文本处理要求。

Unicode编码系统可以分为编码方式和实现方式两个层次。在编码方式上，分为UCS-2和UCS-4两种方式，UCS-2用两个字节编码，UCS-4用4个字节编码。目前实际应用的统一码版本对应于UCS-2，使用16位的编码空间。一个字符的Unicode编码是确定的，但是在实际传输过程中，由于系统平台的不同以及出于节省空间的目的，实现方式有所差异。Unicode的实现方式称为Unicode转换格式（Unicode Transformation Format），简称为UTF，包括UTF-7、UTF-16、UTF-32，UTF-8等，其中较为常见的为UTF-8。UTF-8的特点是对不同范围的字符使用不同长度的编码，其中0x00～0x7F的字符的UTF-8编码与ASCII编码完全相同。UTF-8编码的最大长度是4个字节，从Unicode到UTF-8的编码方式如表2-3所示。

表2-3 Unicode到UTF-8的编码方式

Unicode 编码（十六进制）	UTF-8 字节流（二进制）
000000 ~ 00007F	0xxxxxxx
000080 ~ 0007FF	110xxxxx 10xxxxxx
000800 ~ 00FFFF	1110xxxx 10xxxxxx 10xxxxxx
010000 ~ 10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Unicode经过几十年的发展，逐渐成为业界标准，许多相关技术都提供Unicode支持，如XML、Java、LDAP、CORBA 3.0等，Python也不例外。更多Unicode的知识读者可以查看<http://www.unicode.org/>。

在了解完Unicode的背景知识之后再来看看Python中处理中文字符经常会遇见的以下几个问题：

示例一 读出文件的内容显示为乱码。

```
filehandle = open("test.txt",'r')
print filehandle.read()
filehandle.close()
```

其中，文件test.txt中的内容为“python中文测试”，文件以UTF-8的形式保存。

运行程序结果如下：

```
python
涓
构嫻嫻疾
```

示例二 当Python源文件中包含中文字符的时候抛出SyntaxError异常。

unicodetest.py文件的内容如下：

```
s = "python
中文测试"
print s
```

上述程序运行时报错如下：

```
File "unicodetest.py", line 1
SyntaxError: Non-ASCII character '\xd6' in file unicodetest.py on line 1, but no
encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

示例三 普通字符和Unicode进行字符串连接的时候抛出UnicodeDecodeError异常。

```
# coding=utf-8
s = "
中文测试"+ u"Chinese Test"
print s
```

程序运行抛出异常如下:

```
Traceback (most recent call last):
  File "tests.py", line 2, in <module>
    s = "
中文测试"+ u"Chinese Test"
UnicodeDecodeError: 'ascii' codec can't decode byte 0xd6 in position 0: ordinal
not in range(128)
```

来一一分析上面例子产生错误的原因以及如何在不同的编码和Unicode之间进行转换。

示例一分析：读入的文件test.txt用UTF-8编码形式保存，但是Windows的本地默认编码是CP936，在Windows系统中它被映射为GBK编码，所以当在控制台上直接显示UTF-8字符的时候，这两种编码并不兼容，以UTF-8形式表示的编码在GBK编码中被解释成为其他的符号，由此便产生了乱码。通过上面的背景知识了解到Unicode为不同语言设置了唯一的二进制表示形式，可以轻易地解决不同字符集之间的字符映射问题，因此要解决示例一的乱码问题可以使用Unicode作为中间介质来完成转换。首先需要对读入的字符用UTF-8进行解码，然后再用GBK进行编码。修改后的结果如下：

```
filehandle = open("test.txt",'r')
print (filehandle.read().decode("utf-8")).encode("gbk") .....
①
filehandle.close()
```

输出为:

```
python
中文测试
```

代码标注①处分别使用了decode()和encode()方法，这两个方法的作用分别是对字符串进行解码和编码。其中decode()方法将其他编码对应的字符串解码成Unicode，而encode()方法将Unicode编码转换为另一种编码，Unicode作为转换过程中的中间编码。decode()和encode()方法的函数形式如下：

```
str.decode([
    编码参数[,
    错误处理])
str.encode([
    编码参数[,
    错误处理])
```

常见的编码参数如表2-4所示。

表2-4 常见编码参数

编码参数	描 述
'ascii'	7 位 ASCII 码
'latin-1' or 'iso-8859-1'	ISO 8859-1, Latin-1
'utf-8'	8 位可变长度编码
'utf-16'	16 位可变长度编码
'utf-26-le'	UFT-16, little-endian 编码
'utf-16-be'	UFT-16, big-endian 编码
'unicode-escape'	与 unicode 文字 u'string' 相同
'raw-unicode-escape'	与原始 Unicode 文字 ur'string' 相同

错误处理参数有以下3种常用方式：

- 1) strict': 默认处理方式，编码错误抛出UnicodeError异常。
- 2) ignore'忽略不可转换字符。
- 3) replace'将不可转换字符用?代替。

对于A、B两种编码系统，两者之间的相互转换示意图如图2-1所示。

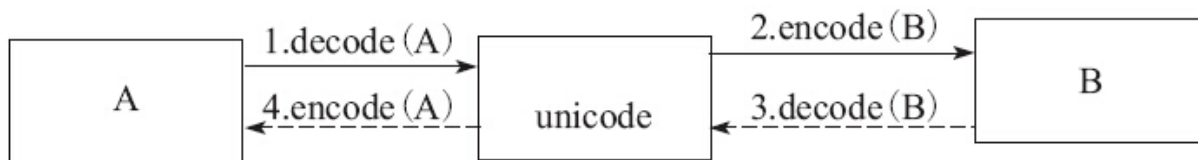


图2-1 编码转换示意图

标注①处`filehandle.read()`读出来的字符串是用UTF-8表示的，也就是A表示为UTF-8，使用`decode()`方法解码得到Unicode，对应上图箭头1所示过程：`filehandle.read().decode("utf-8")`。然后再使用`encode`方法将Unicode转换为为GBK的表示形式，如果`unicodestr=filehandle.read().decode("utf-8")`的话，则`unicodestr.encode("gbk")`表示箭头2所示过程。所以从A到B对应的代码为：

```
(filehandle.read().decode("utf-8")).encode("gbk")
```

提醒：上面的例子在某些有些情况下（如test.txt是用Notepad软件以UTF-8编码形式保存）可能还会出现如下异常：

```
print (filehandle.read().decode("utf-8")).encode("gbk")
UnicodeEncodeError: 'gbk' codec can't encode character u'\uffeff' in position 0:
illegal multibyte sequence
```

这是因为有些软件在保存UTF-8编码的时候，会在文件最开始的地方插入不可见的字符BOM（0xEF 0xBB 0xBF，即BOM），这些不可见字符无法被正确的解析，而利用`codecs`模块可以方便地处理这种问题。

```
import codecs
content = open("test.txt", 'r').read()
filehandle.close()
if content[:3] == codecs.BOM_UTF8:#
    如果存在BOM
    字符则去掉
content = content[3:]
print content.decode("utf-8")
```

关于BOM:

Unicode存储有字节序的问题，例如“汉”字的Unicode编码是0X6C49，如果将6C写在前面，则为big endian，将49写在前面则成为little endian。UTF-16以两个字节为编码单元，在字符的传送过程中，为了标明字节的顺序，Unicode规范中推荐使用BOM（Byte Order Mark）：即在UCS编码中用一个叫做ZERO WIDTH NO-BREAK SPACE的字符，它的编码是FEFF（该编码在UCS中不存在对应的字符），UCS规范建议在传输字节流前，先传输字符ZERO WIDTH NO-BREAK SPACE。这样如果接收者收到FEFF，就表明这个字节流是Big-Endian的；如果收到FFFE，就表明这个字节流是Little-Endian的。UTF-8使用字节来编码，一般不需要BOM来表明字节顺序，但可以用BOM来表明编码方式。字符ZERO WIDTH NO-BREAK SPACE的UTF-8编码是EF BB BF。所以如果接收者收到以EF BB BF开头的字节流，就知道这是UTF-8编码了。

示例二分析：Python中默认的编码是ASCII编码（这点可以通过sys.getdefaultencoding()来验证），所以unicodetest.py文件是以ASCII形式保存的，s是包含中文字符的普通字符串。当调用print方法输出的时候会隐式地进行从ASCII到系统默认编码（Windows上为CP936）的转换，中文字符并不是ASCII字符，而此时源文件中又未指定其他编码方式，Python解释器并不知道如何正确处理这种情况，便会抛出异常：SyntaxError: Non-ASCII character '\xd6' in file unicodetest.py on line 1。因此，要避免这种错误需要在源文件中进行编码声明，声明可用正则表达式

"coding[:]=\s*([-\w.]*)"表示。一般来说进行源文件编码声明有以下3种方式:

第一种声明方式:

```
# coding=<encoding name>
```

第二种声明方式:

```
#!/usr/bin/python  
# -*- coding: <encoding name> -*-
```

第三种声明方式:

```
#!/usr/bin/python  
# vim: set fileencoding=<encoding name> :
```

示例二在源文件头中加入编码声明# coding=utf-8便可解决问题。

示例三分析: 使用+操作符来进行字符串的连接时, +左边为中文字符串, 类型为str, 右边为Unicode字符串。当两种类型的字符串连接的时候, Python将左边的中文字符串转换为Unicode再与右边的Unicode字符串做连接, 将str转换为Unicode时使用系统默认的ASCII编码对字符串进行解码, 但由于“中文测试”的ASCII编码为\x06\xd0\xce\x04\xb2\xe2\xca\xd4, 其中“中”字的编码\x06对应的值为214。当编码值在0~127的时候Unicode和ASCII是兼容的, 转换不会有什么问题, 但当其值大于128的时候, ASCII编码便不能正确处理这种情况, 因而抛出UnicodeDecodeError异常。解决上面的问题有以下两种思路:

1) 指定str转为Unicode时的编码方式。

```
# coding=utf-8  
s = "  
中文测试".decode('gbk') + u"Chinese Test"
```

2) 将Unicode字符串进行UTF-8编码。

```
s = "
中文测试"+ u"Chinese Test".encode("utf-8")
```

Unicode提供了不同编码系统之间字符转换的桥梁，要避免令人头疼的乱码或者避免UnicodeDecodeError以及UnicodeEncodeError等错误，需要弄清楚字符所采用的编码方式以及正确的解码方法。对于中文字符，为了做到不同系统之间的兼容，建议直接使用Unicode表示方式。Python2.6之后可以通过import unicode_literals自动将定义的普通字符识别为Unicode字符串，这样字符串的行为将和Python3中保持一致。

```
>>> from __future__ import unicode_literals
>>> s = "
中文测试"
>>> s
u'\u4e2d\u6587\u6d4b\u8bd5'
```

建议18：构建合理的包层次来管理module

我们知道，本质上每一个Python文件都是一个模块，使用模块可以增强代码的可维护性和可重用性。但显然在大的项目中将所有的Python文件放在一个目录下并不是一个值得推荐的做法，我们需要合理地组织项目的层次来管理模块，这就是包（Package）发挥功效的地方了。

什么是包呢？简单说包即是目录，但与普通目录不同，它除了包含常规的Python文件（也就是模块）以外，还包含一个__init__.py文件，同时它允许嵌套。包结构如下：

```
Package/ __init__.py
        Module1.py
        Module2.py
        Subpackage/ __init__.py
                    Module1.py
                    Module2.py
```

包中的模块可以通过“.”访问符进行访问，即“包名.模块名”。如上述嵌套结构中访问Package目录下的Module1可以使用Package.Module1，而访问Subpackage中的Module1则可以使用Package.Subpackage.Module1。包中的模块同样可以被导入其他模块中。有以下几种导入方法：

1) 直接导入一个包，具体如下：

```
import Package
```

2) 导入子模块或子包，包嵌套的情况下可以进行嵌套导入，具体如下：

```
from Package import Module1
import Package.Module1
from Package import Subpackage
import Package.Subpackage
From Package.Subpackage import Module1
import Package.Subpackage.Module1
```

前面提到在包对应的目录下包含有__init__.py文件，那么这个文件的作用是什么呢？它最明显的作用就是使包和普通目录区分；其次可以在该文件中申明模块级别的import语句从而使其变成包级别可见。上例所示的结构中，如果要import包Package下Module1中的类Test，当__init__.py文件为空的时候需要使用完整的路径来申明import语句：

```
from Package.Module1 import Test
```

但如果在__init__.py文件中添加from Module1 import Test语句，则可以直接使用from Package import Test来导入类Test。需要注意的是，如果__init__.py文件为空，当意图使用from Package import *将包Package中所有的模块导入当前名字空间时并不能使得导入的模块生效，这是因为不同平台间的文件的命名规则不同，Python解释器并不能正确判定模块在对应的平台该如何导入，因此它仅仅执行__init__.py文件，如果要控制模块的导入，则需要对__init__.py文件做修改。

__init__.py文件还有一个作用就是通过在该文件中定义__all__变量，控制需要导入的子包或者模块。在上例的Package目录下的__init__.py文件中添加：

```
__all__ = ['Module1', 'Module2', 'Subpackage']
```

之后再运行`from Package import *`，可以看到`__all__` 变量中定义的模块和包被导入当前名字空间。

```
>>> from Package import *
>>> dir()
['Module1', 'Module2', 'Subpackage', '__builtins__', '__doc__', '__name__',
 '__package__']
```

包的使用能够带来以下便利：

- 合理组织代码，便于维护和使用。通过将关系密切的模块组织成一个包，使项目结构更为完善和合理，从而增强代码的可维护性和实用性。以下是一个可供参考的Python项目结构：

```
ProjectName/
|---README
|   |---LICENSE
|   |---setup.py
|   |---requirements.txt
|   |---sample/
|       |---__init__.py
|       |---core.py
|       |---helpers.py
|   |---docs/
|       |---conf.py
|       |---index.rst
|   |---bin/
|   |---package/
|       |---__init__.py
|       |---subpackage/
|       |---.....
|   |---tests/
|       |---test_basic.py
|       |---test_advanced.py
```

- 能够有效地避免名称空间冲突。使用`from Package import Module2` 可以将Module2导入当前局部名字空间，访问的时候不再需要加入包名。看下面这个例子：

```
>>> from Package import Module2
>>> Module2.Hi()
Hi from Package Module1
```

上述代码中Subpackage中也包含Module2，当使用from... import... 导入的时候，生效的是Subpackage的Module2。结果如下：

```
>>> from Package.Subpackage import Module2
>>> Module2.Hi()
Hi from Subpackage Module2
```

如果模块包含的属性和方法存在同名冲突，使用import module可以有效地避免名称冲突。在嵌套的包结构中，每一个模块都以其所在的完整路径作为其前缀，因此，即使名称一样，但由于模块所对应的其前缀不同，因此不会产生冲突。

```
>>> import Package.Module2
>>> Package.Module2.Hi()
Hi from Package Module1
>>> import Package.Subpackage.Module2
>>> Package.Subpackage.Module2.Hi()
Hi from Subpackage Module2
```

注意：本节所说的包与后文中谈到的软件包不同，这里的包的概念仅限于包含一个或一系列Python文件（模块）的文件夹（目录），它的作用是合理组织代码，便于维护和使用，并避免命名冲突。

第3章 基础语法

Python中常见的基本数据类型有数字、字符串、列表、字典、集合、元组等，常见语法有条件、循环、函数、列表解析等。它们两者组合起来便构成了Python程序的基本要素，可以称之为基础语法。本章我们主要从语法层面阐述一些使用技巧和注意事项。

建议19：有节制地使用from...import语句

Python提供了3种方式来引入外部模块：`import`语句、`from...import...`及`__import__`函数。其中较为常见的为前面两种，而`__import__`函数与`import`语句类似，不同点在于前者显式地将模块的名称作为字符串传递并赋值给命名空间的变量。

在使用`import`的时候注意以下几点：

- 一般情况下尽量优先使用`import a`形式，如访问B时需要使用`a.B`的形式。
- 有节制地使用`from a import B`形式，可以直接访问B。
- 尽量避免使用`from a import *`，因为这会污染命名空间，并且无法清晰地表示导入了哪些对象。

为什么在使用`import`的时候要注意以上几点呢？在回答这个问题之前先来简单了解一下Python的`import`机制。Python在初始化运行环境的时候会预先加载一批内建模块到内存中，这些模块相关的信息被存放在`sys.modules`中。读者导入`sys`模块后在Python解释器中输入`sys.modules.items()`便可显示所有预加载模块的相关信息。当加载一个模块的时候，解释器实际上要完成以下动作：

- 1) 在`sys.modules`中进行搜索看看该模块是否已经存在，如果存在，则将其导入到当前局部命名空间，加载结束。
- 2) 如果在`sys.modules`中找不到对应模块的名称，则为需要导入的模块创建一个字典对象，并将该对象信息插入`sys.modules`中。

3) 加载前确认是否需要对模块对应的文件进行编译，如果需要则先进行编译。

4) 执行动态加载，在当前模块的命名空间中执行编译后的字节码，并将其中所有的对象放入模块对应的字典中。

我们以用户自定义的模块为例来看看`sys.modules`和当前局部命名空间发生的变化。在Python的安装目录下创建一个简单的模块`testmodule.py`:

```
a = 1
b = 'a'
print "testing module import"
```

我们知道用户模块未加载之前，`sys.modules`中并不存在相关信息。那么进行`import testmodule`操作会发生什么情况呢？

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
>>> import testmodule
testing module import
>>> dir()
①import testmodule
之后局部命名空间发生变化
['__builtins__', '__doc__', '__name__', '__package__', 'sys', 'testmodule']
>>> 'testmodule' in sys.modules.keys()
True
>>> id(testmodule)
35776304
>>> id(sys.modules['testmodule'])
35776304
>>> dir(testmodule)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'a', 'b']
>>> sys.modules['testmodule'].__dict__.keys()
['a', 'b', '__builtins__', '__file__', '__package__', '__name__', '__doc__',]
```

从输出结果可以看出，对于用户定义的模块，`import`机制会创建一个新的`module`将其加入当前的局部命名空间中，与此同时，`sys.modules`也加入了该模块的相关信息。但从它们的`id`输出结果可以看出，本质上是引用同一个对象。同时会发现`testmodule.py`所在的目

录下多了一个.pyc的文件，该文件为解释器生成的模块相对应的字节码，从import之后的输出“testing module import”可以看出模块同时被执行，而a和b被写入testmodule所对应的字典信息中。

需要注意的是，直接使用import和使用from a import B形式这两者之间存在一定的差异，后者直接将B暴露于当前局部空间，而将a加载到sys.modules集合。

```
>>> import sys
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
>>> from testmodule import a
testing module import
>>> dir()
①使用from.....import.....
之后命名空间发生的变化
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'sys']
>>> sys.modules['testmodule']
<module 'testmodule' from 'testmodule.pyc'>
>>> id(sys.modules['testmodule'])
36562576
>>> id(a)
31697400
>>> id(sys.modules['a'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    KeyError: 'a'
```

了解完import机制，我们再来看看对于from a import ...无节制的使用会带来什么问题。

(1) 命名空间的冲突

来看一个例子。假设有如下3个文件：a.py，b.py及importtest.py，其中a和b都定义了add()函数，当在import test文件中同时采用from...import...的形式导入add的时候，import test中起作用的到底是哪一个函数呢？

文件a.py如下：

```
def add():  
    print "add in module A"
```

文件b.py如下:

```
def add():  
    print "math in module B"
```

文件importtest.py如下:

```
from a import add  
from b import add  
if __name__ == '__main__':  
    math()
```

从程序的输出“add in module B”可以看出实际起作用的是最近导入的add(), 它完全覆盖了当前命名空间之前从a中导入的add()。在项目中, 特别是大型项目中频繁地使用from a import ...的形式会增加命名空间冲突的概率从而导致出现无法预料的问题。因此需要有节制地使用from...import语句。一般来说在非常明确不会造成命名冲突的前提下, 以下几种情况下可以考虑使用from...import语句:

- 1) 当只需要导入部分属性或方法时。
- 2) 模块中的这些属性和方法访问频率较高导致使用“模块名.名称”的形式进行访问过于烦琐时。
- 3) 模块的文档明确说明需要使用from...import形式, 导入的是一个包下面的子模块, 且使用from...import形式能够更为简单和便利时。如使用from io.drivers import zip要比使用import io.drivers.zip更方便。

(2) 循环嵌套导入的问题

先来看下面的例子：

```
c1.py:
from c2 import g
def x():
    Pass
c2.py:
from c1 import x
def g():
    Pass
```

无论运行上面哪一个文件都会抛出**ImportError**异常。这是因为在执行**c1.py**的加载过程中，需要创建新的模块对象**c1**然后执行**c1.py**所对应的字节码。此时遇到语句**from c2 import g**，而**c2**在**sys.modules**也不存在，故此时创建与**c2**对应的模块对象并执行**c2.py**所对应的字节码。当遇到**c2**中的语句**from c1 import x**时，由于**c1**已经存在，于是便去其对应的字典中查找**g**，但**c1**模块对象虽然创建但初始化的过程并未完成，因此其对应的字典中并不存在**g**对象，此时便抛出**ImportError: cannot import name g**异常。而解决循环嵌套导入问题的一个方法是直接使用**import**语句。读者可以自行验证。

建议20： 优先使用absolute import来导入模块

假设有如下文件结构，其中app/sub1/string.py中定义了一个lower()方法，那么当在mod1.py中import string之后再使用string.lower()方法时，到底引用的是sub1/string.py中的lower()方法，还是Python标准库中string里面的lower()方法呢？

```
app/  
  __init__.py  
  sub1/  
    __init__.py  
    mod1.py  
    string.py  
  sub2/  
    __init__.py  
    mod2.py
```

从程序的输出会发现，它引用的是app/sub1/string.py中的lower()方法。显然解释器默认先从当前目录下搜索对应的模块，当搜到string.py的时候便停止搜索进行动态加载。那么，如果要使用Python自带的string模块中的方法，该怎么实现呢？这就涉及absolute import和relative import相关的话题了。

在Python2.4以前默认为隐式的relative import，局部范围的模块将覆盖同名的全局范围的模块。如果要使用标注库中同名的模块，你不得不去深入考察sys.modules一番，显然这并不是是一种非常友好的做法。Python2.5中后虽然默认的仍然是relative import，但它为absolute import提供了一种新的机制，在模块中使用from __future__ import absolute_import 语句进行说明后再进行导入。同时它还通过点号提供了一种显式进行relative import的方法，“.”表示当前目录，“..”表示当前

目录的上一层目录。例如想在mod1.py中导入string.py，可以使用from . import string，其中mod1所在的包层次结构为app.sub1.mod1，“.”表示app.sub1；如果想导入sub2/mod2.py可以使用from ..sub2 import mod2，“..”代表的是app。

但事情是不是就此结束了呢？远不止，使用显式relative import之后再运行程序一不小心你就有可能遇到这种错误“ValueError: Attempted relative import in non-package”。这是什么原因呢？这个问题产生的原因在于relative import使用模块的__name__属性来决定当前模块在包层次结构中的位置，如果当前的模块名称中不包含任何包的信息，那么它将默认为模块在包的顶层位置，而不管模块在文件系统中的实际位置。而在relative import的情形下，__name__会随着文件加载方式的不同而发生改变，上例中如在目录app/sub1/下运行Python mod1.py，会发现模块的__name__为__main__，但如果在目录app/sub1/下运行Python-m mod1.py，会发现__name__变为mod1。其中-m的作用是使得一个模块像脚本一样运行。而无论以何种方式加载，当在包的内部运行脚本的时候，包相关的结构信息都会丢失，默认当前脚本所在的位置为模块在包中的顶层位置，因此便会抛出异常。如果确实需要将模块当作脚本一样运行，解决方法之一是在包的顶层目录中加入参数-m运行该脚本，上例中如果要运行脚本mod1.py可以在app所在的目录的位置输入Python -m app.sub1.mod1。另一个解决这个问题的方法是利用Python2.6在模块中引入的__package__属性，设置__package__之后，解释器会根据__package__和__name__的值来确定包的层次结构。上面的例子中如果将mod1.py修改为以下形式便不会出现在包结构内运行模块对应的脚本时出错的情况了。

```
if __name__ == "__main__" and __package__ is None:
    import sys
    import os.path
    sys.path[0] = os.path.abspath("../..")
    print sys.path[0]
    import app.sub1
```

```
__package__ = str('app.sub1')  
from . import string
```

相比于**absolute import**，**relative import**在实际应用中反馈的问题较多，因此推荐优先使用**absolute import**。 **absolute import**可读性和出现问题后的可跟踪性都更好。当项目的包层次结构较为复杂的时候，显式**relative import**也是可以接受的，由于命名冲突的原因以及语义模糊等原因，不推荐使用隐式的**relative import**，并且它在Python3中已经被移除。

建议21: `i+=1`不等于`++i`

对于对Python语言的每个细节了解得不是那么清楚，而恰好又有其他语言背景的开发人员，很有可能写出如下类似的代码：

```
i = 0
mylist = [1,2,3,4,5,6]
while i < len(mylist):
    print mylist[i]
    ++i
```

运行这段代码会有什么问题？也许你会说：抛出语法错误。能说出这个答案的至少知道Python中是不支持`++i`操作的。但输出果真如此吗？非也，这段程序不会抛出任何语法错误，却会无限循环地输出1。原因是什么呢？因为Python解释器会将`++i`操作解释为`+(+i)`，其中`+`表示正数符号。对于`--i`操作也是类似。

```
>>> +1
1
>>> ++1
1
>>> ++++1
1
>>> -2
-2
>>> --2 #
负负得正
2
>>> -----2
-2
>>>
```

因此你需要明白`++i`在Python中语法上是合法的，但并不是我们理解的通常意义上的自增操作。

建议22：使用with自动关闭资源

来做个简单的试验，观察一下发生的现象。在Python解释器中输入下面两行代码，会有什么情况发生呢？

```
>>> f = open('test.txt', 'w')
>>> f.write("test")
```

答案是：在解释器所在的目录下生成了一个文件test.txt，并且在里面写入了字符串test，对吗？事实真相是：的确生成了一个文件，但其内容为空，并没有写入任何字符串。这个一个简单得不能再简单的问题，相信不用多说你已经知道症结所在了。

对文件操作完成后应该立即关闭它们，这是一个常识。我们都知道需要这么做，在很多编程语言中都会强调这个问题，因为打开的文件不仅会占用系统资源，而且可能影响其他程序或者进程的操作，甚至会导致用户期望与实际操作结果不一致。但实际应用中真相往往是：即使我们心中记得这个原则，但仍然可能会忘记关闭它。为什么？因为编程人员会把更多的精力和注意力放在对具体文件内容的操作和处理上；或者设计的正常流程是处理完毕关闭文件，但结果程序执行过程中发生了异常导致关闭文件的代码没有被执行到。也许你会说，还有try..finally块。对！这是一种比较古老的方法，但Python提供了一种更为简单的解决方案：with语句。with语句的语法为：

```
with
表达式  [ as
目标]
:
代码块
```

with语句支持嵌套，支持多个**with**子句，它们两者可以相互转换。“**with** **expr1** as **e1** , **expr2** as **e2**”与下面的嵌套形式等价：

```
with expr1 as e1:  
    with expr2 as e2:
```

with语句的使用非常简单，本节开头的例子改用**with**语句能够保证当写操作执行完毕后自动关闭文件。

```
>>> with open('test.txt','w') as f:  
...     f.write("test")  
...
```

with语句可以在代码块执行完毕后还原进入该代码块时的现场。包含有**with**语句的代码块的执行过程如下：

- 1) 计算表达式的值，返回一个上下文管理器对象。
- 2) 加载上下文管理器对象的__exit__()方法以备后用。
- 3) 调用上下文管理器对象的__enter__()方法。
- 4) 如果**with**语句中设置了目标对象，则将__enter__()方法的返回值赋值给目标对象。
- 5) 执行**with**中的代码块。
- 6) 如果步骤5中代码正常结束，调用上下文管理器对象的__exit__()方法，其返回值直接忽略。
- 7) 如果步骤5中代码执行过程中发生异常，调用上下文管理器对象的__exit__()方法，并将异常类型、值及traceback信息作为参数传递给

`__exit__()`方法。如果`__exit__()`返回值为`false`，则异常会被重新抛出；如果其返回值为`true`，异常被挂起，程序继续执行。

在文件处理时使用`with`的好处在于无论程序以何种方式跳出`with`块，总能保证文件被正确关闭。实际上它不仅仅针对文件处理，针对其他情景同样可以实现运行时环境的清理与还原，如多线程编程中的锁对象的管理。`with`的神奇实际得益于一个称为上下文管理器

（`context manager`）的东西，它用来创建一个运行时的环境。上下文管理器是这样一个对象：它定义程序运行时需要建立的上下文，处理程序的进入和退出，实现了上下文管理协议，即在对象中定义`__enter__()`和`__exit__()`方法。其中：

- `__enter__()`：进入运行时的上下文，返回运行时上下文相关的对象，`with`语句中会将这个返回值绑定到目标对象。如上面的例子中会将文件对象本身返回并绑定到目标`f`。

- `__exit__(exception_type,exception_value,traceback)`：退出运行时的上下文，定义在块执行（或终止）之后上下文管理器应该做什么。它可以处理异常、清理现场或者处理`with`块中语句执行完成之后需要处理的动作。

实际上任何实现了上下文协议的对象都可以称为一个上下文管理器，文件也是实现了这个协议的上下文管理器，它们都能够与`with`语句兼容。文件对象的`__enter__`和`__exit__`属性如下：

```
>>> f.__enter__
<built-in method __enter__ of file object at 0x029F0700>
>>> f.__exit__
<built-in method __exit__ of file object at 0x029F0700>
```

用户也可以定义自己的上下文管理器来控制程序的运行，只需要实现上下文协议便能够与`with`语句一起使用。

```
>>> class MyContextManager(object):
...     def __enter__(self):#
实现__enter__
方法
...         print "entering..."
...     def __exit__(self,exception_type, exception_value, traceback):
...         print "leaving..."
...         if exception_type is None:
...             print "no exceptions!"
...             return False
...         elif exception_type is ValueError:
...             print "value error!!!"
...             return True
...         else:
...             print "other error"
...             return True
...
>>>
>>> with MyContextManager():
...     print "Testing..."
...     raise(ValueError)
...
entering...
Testing...
leaving...
value error!!!
>>>
>>> with MyContextManager():
...     print "Testing..."
...
entering...
Testing...
leaving...
no exceptions!
>>>
```

因为上下文管理器主要作用于资源共享，因此在实际应用中__enter__()和__exit__()方法基本用于资源分配以及释放相关的工作，如打开/关闭文件、异常处理、断开流的连接、锁分配等。为了更好地辅助上下文管理，Python还提供了contextlib模块，该模块是通过Generator实现的，contextlib中的contextmanager作为装饰器来提供一种针对函数级别的上下文管理机制，可以直接作用于函数/对象而不用去关心__enter__()和__exit__()方法的具体实现。关于contextlib更多内容读者可以参考网页<http://docs.python.org/2/library/contextlib.html>。

建议23：使用else子句简化循环（异常处理）

有其他编程语言经验的程序员接触到Python时，对于它无所不在的else往往感到非常惊讶。在Python中，不仅分支语句有else子句，而且循环语句也有，甚至连异常处理也有。首先来看看循环语句中的else，看看它们的语法。

```
while_stmt ::= "while" expression ":" suite
             ["else" ":" suite]
for_stmt  ::= "for" target_list "in" expression_list ":" suite
             ["else" ":" suite]
```

从语法定义中可以看到如果没有["else" ":" suite]这一块，Python的循环语句跟大多数语言并无二致。要谈else子句，必须先从Python从其他语言中借鉴的语义相同的break语句说起，因为else子句提供了隐含的对循环是否由break语句引发循环结束的判断。先来看一个没有应用else子句的例子：

```
def print_prime(n):
    for i in xrange(2, n):
        found = True
        for j in xrange(2, i):
            if i % j == 0:
                found = False
                break
        if found:
            print '%d is a prime number'%i
```

这是一个查找素数的简单实现，可以看到我们借助了一个标志量found来判断是循环结束是不是由break语句引起的。如果对else善加利用，代码可以简洁得多。来看下面的具体实现：

```
def print_prime2(n):
    for i in xrange(2, n):
        for j in xrange(2, i):
            if i % j == 0:
                break
        else:
            print '%d is a prime number'%i
```

当循环“自然”终结（循环条件为假）时`else`从句会被执行一次，而当循环是由`break`语句中断时，`else`子句就不被执行。与`for`语句相似，`while`语句中的`else`子句的语意是一样的：`else`块在循环正常结束和循环条件不成立时被执行。

与C/C++等较为“老土”的语言相比，`else`子句使程序员的生产力和代码的可读性都得到了提高，所以建议大家多使用`else`，让程序变得更加Pythonic。

在Python的异常处理中，也提供了`else`子句语法，这颗“语法糖”的意义跟循环语句中的`else`是相似的：`try`块没有抛出任何异常时，执行`else`块。按惯例先看一下如下语法定义：

```
try_stmt ::= try1_stmt | try2_stmt
try1_stmt ::= "try" ":" suite
              ("except" [expression [("as" | ",") target]] ":" suite)+
              ["else" ":" suite]
              ["finally" ":" suite]
try2_stmt ::= "try" ":" suite
              "finally" ":" suite
```

从`try1_stmt`的定义中可以看到，Python的异常处理中有一种`try-except-else-finally`形式。下面的例子是把数据写入文件中。

```
def save(db, obj):
    try:
        # save attr1
        db.execute('a sql stmt', obj.attr1)
        # save attr2
        db.execute('another sql stmt', obj.attr2)
    except DBError:
        db.rollback()
    else:
        db.commit()
```

如果没有**else**子句，就像前文中关于循环的例子一样，需要引入一个标志变量。

```
def save(db, obj):
    some_error_occurred = False
    try:
        # save attr1
        db.execute('a sql stmt', obj.attr1)
        # save attr2
        db.execute('another sql stmt', obj.attr2)
    except DBError:
        db.rollback()
        some_error_occurred = True
    if not some_error_occurred:
        db.commit()
```

这样代码就变得复杂了。在Python中还有不少语法都是致力于让程序员可以编写更加简明、更接近自然语言语义的代码，比如**in**和**with**语句（将在其他章中讲述相关用法），这也证明充分地学习手册中的**Language Reference**非常有必要。

建议24：遵循异常处理的几点基本原则

现实世界是不完美的，意外和异常会在不经意间发生，从而使我们的生活不得不暂时偏离正常轨道，软件世界也是如此。或因为外部原因，或因为内部原因，程序会在某些条件下产生异常或者错误。为了提高系统的健壮性和用户的友好性，需要一定的机制来处理这种情况。跟其他很多编程语言一样，Python也提供了异常处理机制。Python中常用的异常处理语法是try、except、else、finally，它们可以有多种组合，如try-except（一个或多个），try-except-else；try-finally以及try-except-else-finally等。语法形式如下：

```
try:
    <statements>                # Run this main action first
except <name1>:
    <statements>                #
    当 try
    中发生name1
    的异常时处理
except (name2, name3):
    <statements>                #
    当try
    中发生name2
    或name3
    中的某一个异常的时候处理
except <name4> as <data>:
    <statements>                #
    当 try
    中发生name4
    的异常时处理，并获取对应实例
except:
    <statements>                #
    其他异常发生时处理
else:
    <statements>                #
    没有异常发生时执行
finally:
    <statements>                #
    不管有没有异常发生都会执行
```

最为全面的组合try-except-else-finally异常处理的流程如图3-1所示。

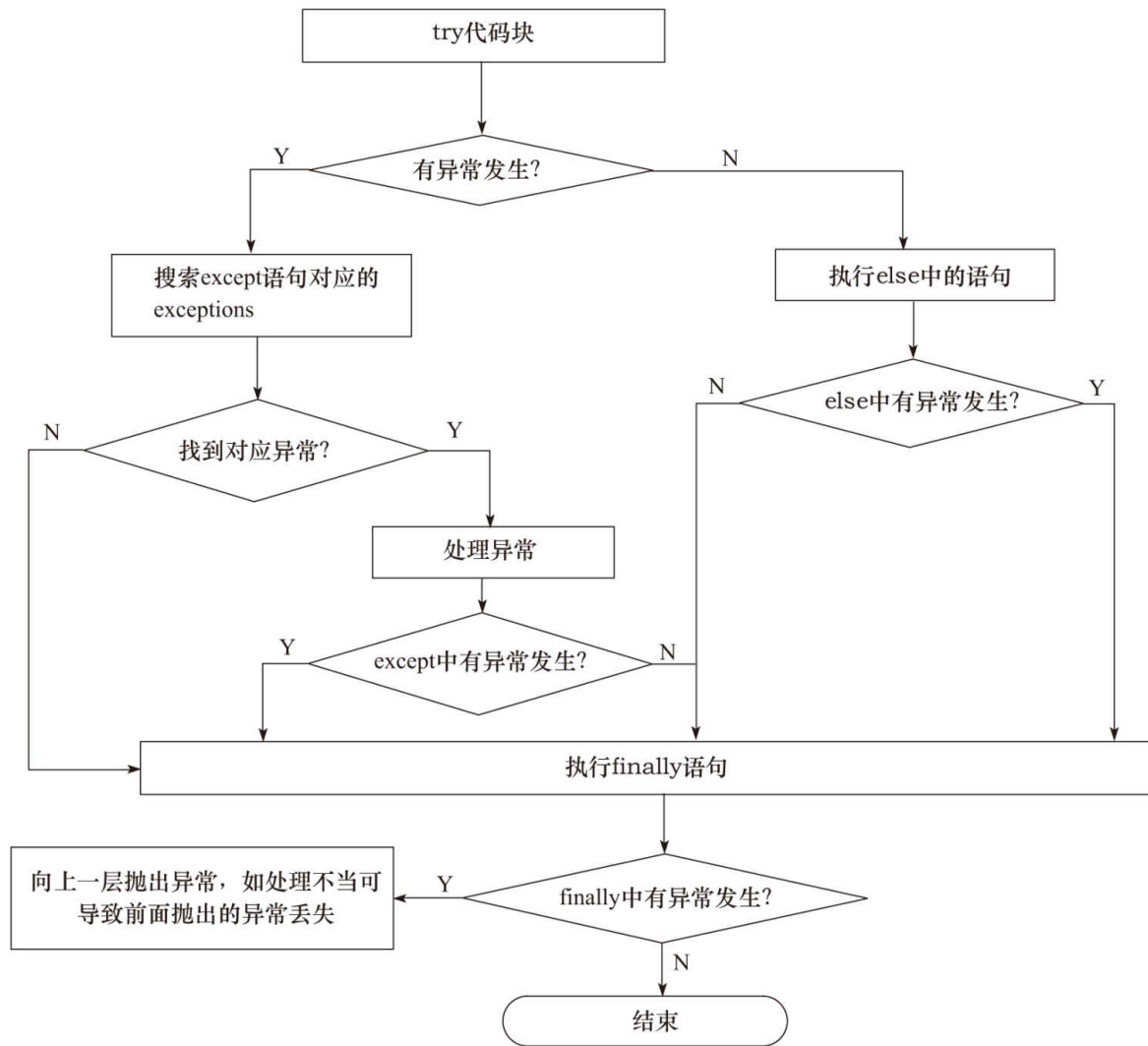


图3-1 异常处理的流程图

异常处理通常需要遵循以下几点基本原则：

1) 注意异常的粒度，不推荐在try中放入过多的代码。异常的粒度是人为划分的，在处理异常的时候最好保持异常粒度的一致性和合理性，同时要避免在try中放入过多的代码，即避免异常粒度过大。在try中放入过多的代码带来的问题是如果程序中抛出异常，将会较难定位，给debug和修复带来不便，因此应尽量只在可能抛出异常的语句块前面放入try语句。

2) 谨慎使用单独的except语句处理所有异常，最好能定位具体的异常。同样也不推荐使用except Exception或者except StandardError来捕获异常。

在try后面单独使用except语句可以捕获所有的异常，从表面上看这似乎是个不错的做法，但实际上会带来什么问题呢？来看以下简单的例子：

```
import sys
try:
    print a
    b =0
    print a/b
except:
    sys.exit("ZeroDivisionError:Can not division zero")
```

程序运行以打印“ZeroDivisionError: Can not division zero”结束，这会让我们以为是发生了除数为零的错误，但实际情况是因为a在使用前并没有定义，程序引发了NameError。而由于单独的except语句的使用，真实的错误往往被掩盖。对上述代码修改如下：

```
import sys
try:
    print a
    b =0
    print a/b
except ZeroDivisionError:
    sys.exit("ZeroDivisionError:Can not division zero")
```

运行程序输出NameError异常如下：

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    print a
NameError: name 'a' is not defined
```

单独使用except会捕获包括SystemExit，KeyboardInterrupt等在内的各种异常，从而掩盖程序真正发生异常的原因，给debug造成一定的迷

惑性。因此需要谨慎使用，最好能在except语句中定位具体的异常。如果在某些情况下不得不使用单独的except语句，最好能够使用raise语句将异常抛出向上层传递。

3) 注意异常捕获的顺序，在合适的层次处理异常。Python中内建异常以类的形式出现，Python2.5后异常被迁移到新式类上，启用了新的所有异常之母的BaseException类，内建异常有一定的继承结构，如UnicodeDecodeError继承自UnicodeError，而其继承链结构为UnicodeDecodeError --> UnicodeError --> ValueError --> Exception --> BaseException，如图3-2所示。

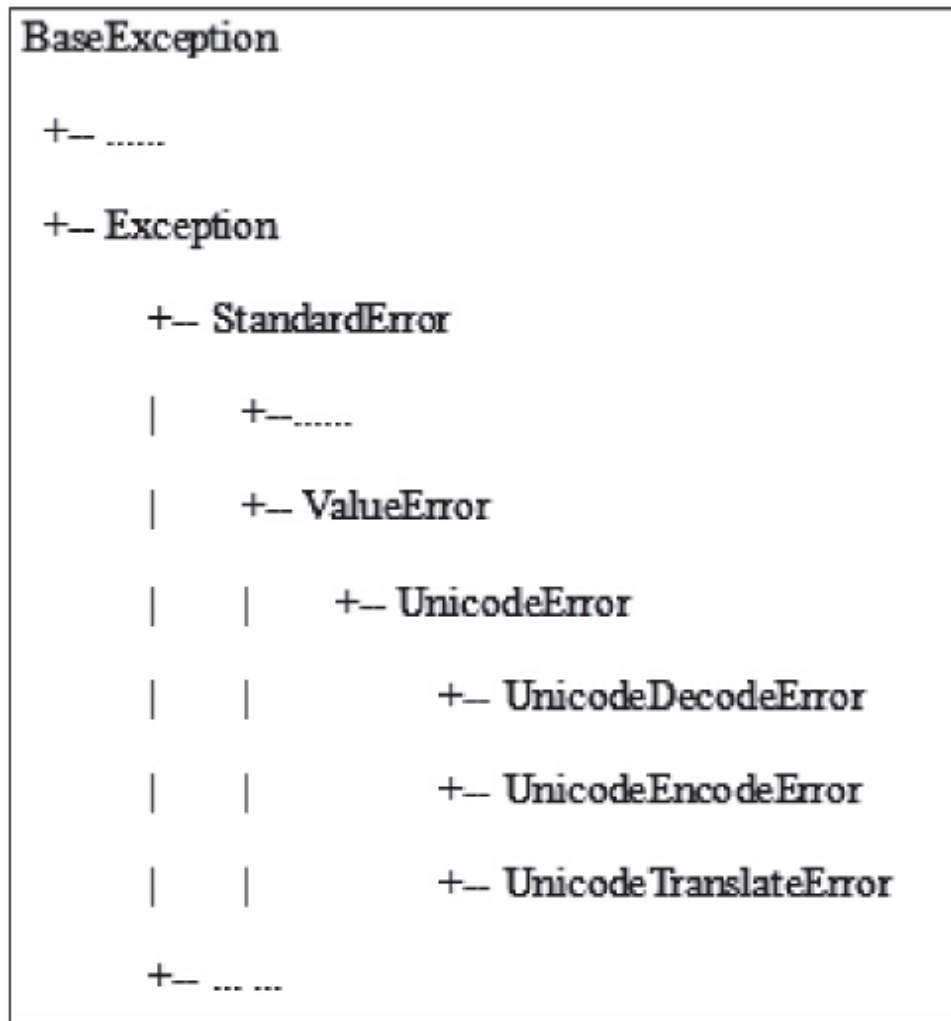


图3-2 UnicodeDecodeError继承结构示意图

用户也可以继承自内建异常构建自己的异常类，从而在内建类的继承结构上进一步延伸。在这种情况下异常捕获的顺序显得非常重要。为了更精确地定位错误发生的原因，推荐的方法是将继承结构中子类异常在前面的except语句中抛出，而父类异常在后面的except语句抛出。这样做的原因是当try块中有异常发生的时候，解释器根据except声明的顺序进行匹配，在第一个匹配的地方便立即处理该异常。如果将层次较高的异常类在前面进行捕获，往往不能精确地定位异常发生的具体位置。如下例中ValueError声明在前而UnicodeDecodeError在后，当抛出UnicodeDecodeError异常的时候，由于它是ValueError的子类，在ValueError处便直接被捕获了，打印出消息“ValueError occurred”，而真正的异常UnicodeDecodeError却被悄然掩盖。

```
>>> try:
...     raise UnicodeDecodeError("pdfdocencoding","a",2,-1,"not support decoding")
... except ValueError:#ValueError
为UnicodeDecodeError
的父类，捕获异常时却在前面
...     print "ValueError occurred"
... except UnicodeDecodeError,e:
...     print e
...
ValueError occurred
>>>
```

因此，异常捕获的顺序非常重要，同时异常应该在适当的位置被处理，一个原则就是如果异常能够在被捕获的位置被处理，那么应该及时处理，不能处理也应该以合适的方式向上层抛出。遇到异常不论好歹就向上层抛出是非常不明智的。向上层传递的时候需要警惕异常被丢失的情况，可以使用不带参数的raise来传递。

```
try:
    some_code()
except:
    revert_stuff()
    raise
```

4) 使用更为友好的异常信息，遵守异常参数的规范。软件最终是为用户服务的，当异常发生的时候，异常信息清晰友好与否直接关系到用户体验。通常来说有两类异常阅读者：使用软件的人和开发软件的人，即用户和开发者。对于用户来说关注更多的是业务。先来看一段异常信息：

```
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print ItemPriceTable['a']
KeyError: 'a'
```

如果将这段信息给一个没有软件编程背景的人看，他肯定会觉得如读天书一般，对于用户这并不是一种较为友好的做法，在面对用户的时候异常信息应该以较为清晰和明确的方式显示出来。如下例中当查找一个不在列表的水果价格的时候给出相关的提示信息会比直接抛出**KeyError**信息要友好得多。

```
import sys
import traceback
ItemPriceTable = {"apple":'3.5',"orange":'4',"cheery":'20',"mango":'8'}
def getprice(itemname):
    try:
        price = ItemPriceTable[itemname]
        return price
    except KeyError:
        print "%s can not find in the price table,you should input another kind of fruit."
        % sys.exc_value #
显示异常相关的提示信息
while 1:
    itemname = raw_input("Enter the fruit name to get the price or press x to exit: ")
    if itemname == "x":
        break
    price = getprice(itemname)
    if price != None:
        print "%s's price is $%s/kg" % (itemname, price)
```

此外，如果内建异常类不能满足需求，用户可以在继承内建异常的基础上针对特定的业务逻辑定义自己的异常类。但无论是内建异常

类，还是用户定义的异常类，在传递异常参数的时候都需要遵守异常参数规范。

建议25：避免finally中可能发生的陷阱

无论try语句中是否有异常抛出，finally语句总会被执行。由于这个特性，finally语句经常被用来做一些清理工作，如打开一个文件，抛出异常后在finally语句中对文件句柄进行关闭等。

但使用finally时，也要特别小心一些陷阱。先来看以下例子：

```
def FinallyTest():
    print 'i am starting-----'
    while True:
        try:
            print "I am running"
            raise IndexError("r")#
抛出异常IndexError
异常
        except NameError,e:
            print 'NameError happended %s',e
            break
        finally:
            print 'finally executed'
            break #finally
语句中有break
语句
FinallyTest()
```

上述程序输出结果为：

```
i am starting-----
I am running
finally executed
```

上面的例子中try代码块抛出了IndexError异常，但在except块却没有对应的异常声明。按常理该异常会向上层抛出，可是程序输出却没有提示任何异常发生，IndexError异常被丢失。这是什么原因呢？当try块中发生异常的时候，如果在except语句中找不到对应的异常处理，异常将会被临时保存起来，当finally执行完毕的时候，临时保存的异常将会再次被抛出，但如果finally语句中产生了新的异常或者执行了

`return`或者`break`语句，那么临时保存的异常将会被丢失，从而导致异常屏蔽。这是`finally`使用时需要小心的第一个陷阱。再来看另外一个例子：

```
def ReturnTest(a):
    try:
        if a <=0:
            raise ValueError("data can not be negative")
        else:
            return a
    except ValueError as e:
        print e
    finally:
        print("The End!")
        return -1
print ReturnTest(0)
print ReturnTest(2)
```

思考一下这里程序`ReturnTest(0)`和`ReturnTest(2)`的返回值是什么？答案是：-1， -1。对于第一个调用`ReturnTest(0)`在抛出`ValueError`异常后直接执行`finally`语句返回值为-1，这点比较容易理解；那么对于第二个调用`ReturnTest(2)`为什么也返回-1呢？这是因为`a>0`，会执行`else`分支，但由于存在`finally`语句，在执行`else`语句的`return a`语句之前会先执行`finally`中的语句，此时由于`finally`语句中有`return -1`，程序直接返回了，所以永远不会返回`a`对应的值2。此为使用`finally`语句需要注意的第二个陷阱。在实际应用程序开发过程中，并不推荐在`finally`中使用`return`语句进行返回，这种处理方式不仅会带来误解而且可能会引起非常严重的错误。

建议26： 深入理解None， 正确判断对象是否为空

在学习Python的过程中，可能曾经有人写过以下代码用来判断变量a是否为空：

```
if a is not None:           #value is not empty
    Do something
else:                       #value is empty
    Do some other thing
```

那么这样写有什么问题呢？先来了解一下Python中哪些形式的数据为空。Python中以下数据会当做空来处理：

- 常量None。
- 常量False。
- 任何形式的数值类型零，如0、0L、0.0、0j。
- 空的序列，如"、()、[]。
- 空的字典，如{}。
- 当用户定义的类中定义了nonzero()方法和len()方法，并且该方法返回整数0或者布尔值False的时候。

其中常量None的特殊性体现在它既不是0、False，也不是空字符串，它就是一个空值对象。其数据类型为NoneType，遵循单例模式，

是唯一的，因而不能创建None对象。所有赋值为None的变量都相等，并且None与任何其他非None的对象比较结果都为False。

```
>>> id(None)
505555980
>>> None == 0
False
#None
不为0
>>> None == False
False
#None
也不是False
>>> None == ""
False
#None
更不是空字符串
>>> a = None
>>> id(a)
505555980
>>> b = None
>>> a == b
True
#
任何赋值为None
的对象都相同
>>> list1 = []
>>> if list1 is not None:
①判断list
是否为空
...     print "list is:",list1
... else:
...     print "list is empty"
...
list is: []
②输出结果表示上面的逻辑认为list
不为空
```

上面的例子中对列表是否为空的判断显然不符合我们的要求，因为除非a被赋值为None，否则else中的语句永远不会被执行。正确的形式如下：

```
if list1:    #value is not empty
③判断list1
是否为空的正确方式
    Do something
else:
    #value is empty
    Do some other thing
```

标注③执行过程中会调用内部方法__nonzero__()来判断变量list1是否为空并返回其结果。下面介绍一下__nonzero__()方法：该内部方法用于对自身对象进行空值测试，返回0/1或True/False。如果一个对象没

有定义该方法，Python将获取__len__()方法调用的结果来进行判断。
__len__()返回值为0则表示为空。如果一个类中既没有定义__len__()方法也没有定义__nonzero__()方法，该类的实例用if判断的结果都为True。

```
def __nonzero__(self): #
    类中实现了 __nonzero__
    方法
    print'testing A.__nonzero__()'
    return True
def __len__(self):
    print "get length"
    return False
if A():#
    该语句执行的时候会自动调用 __nonzero__
    () 方法
    print 'not empty'
else:
    print "empty"
```

程序输出如下：

```
testing A.__nonzero__()
not empty
```

建议27： 连接字符串应优先使用join而不是+

字符串处理在大多数编程语言中都不可避免，字符串的连接也是在编程过程中经常需要面对的问题。Python中的字符串与其他一些程序语言如C++、Java有一些不同，它为不可变对象，一旦创建便不能改变，它的这个特性直接影响到Python中字符串连接的效率。我们首先来看常见的两种字符串连接方法。

1) 使用操作符+连接字符串的方法如下：

```
>>> str1,str2,str3 = 'testing ','string ','concatenation '
>>> str1+str2+str3
'testing string concatenation '
>>>
```

2) 使用join方法连接字符串的方法如下：

```
>>> str1,str2,str3 = 'testing ','string ','concatenation '
>>> ''.join([str1,str2,str3])
'testing string concatenation '
>>>
```

思考这么一个问题：上述两种字符串连接的方法除了使用形式上的不同还有其他区别吗？性能上会不会有所差异呢？来看下面这个测试例子：

```
import timeit
#
生成测试所需要的字符数组
strlist=["it is a long value string will not keep in memory" for n in
range(100000)]
#100000
为字符串连接的数目，下面对应的测试数据，每次需要修改
def join_test():
```

```
        return ''.join(strlist)      #
使用join
方法连接strlist
中的元素并返回字符串
def plus_test():
    result = ''
    for i,v in enumerate(strlist):
        result= result+ v            #
使用+
进行字符串连接
    return result
if __name__ == '__main__':
    jointimer = timeit.Timer("join_test()", "from __main__ import join_test")
    print jointimer.timeit(number = 100)
    plustimer = timeit.Timer("plus_test()", "from __main__ import plus_test")
    print plustimer.timeit(number = 100)
```

给上面的程序传入一组测试参数（测试参数为3，10，100，1000，10000，100000；分别表示每一次测试所要连接的字符串的数量），程序用于测试join_test()和plus_test()这两个方法在字符串连接规模改变时所消耗时间的变化。测试结果记录如表3-1所示。

连接的字符串数量	join_test 运行时间	plus_test 运行时间
3	0.0000389002415462	0.000132909158616
10	0.000126425785025	0.000602548533116
100	0.00033997190268	0.00357801180055
1000	0.00274368266155	0.030371768798
10000	0.0343671477735	0.379505083573
100000	0.441415223204	187.267786021

表3-1 join_test()和plus_test()连接字符串所耗时间记录

表3-1可以用图3-3所示的X-Y图表示，其中X轴表示所要连接的字符串的数量，Y轴表示消耗的时间。

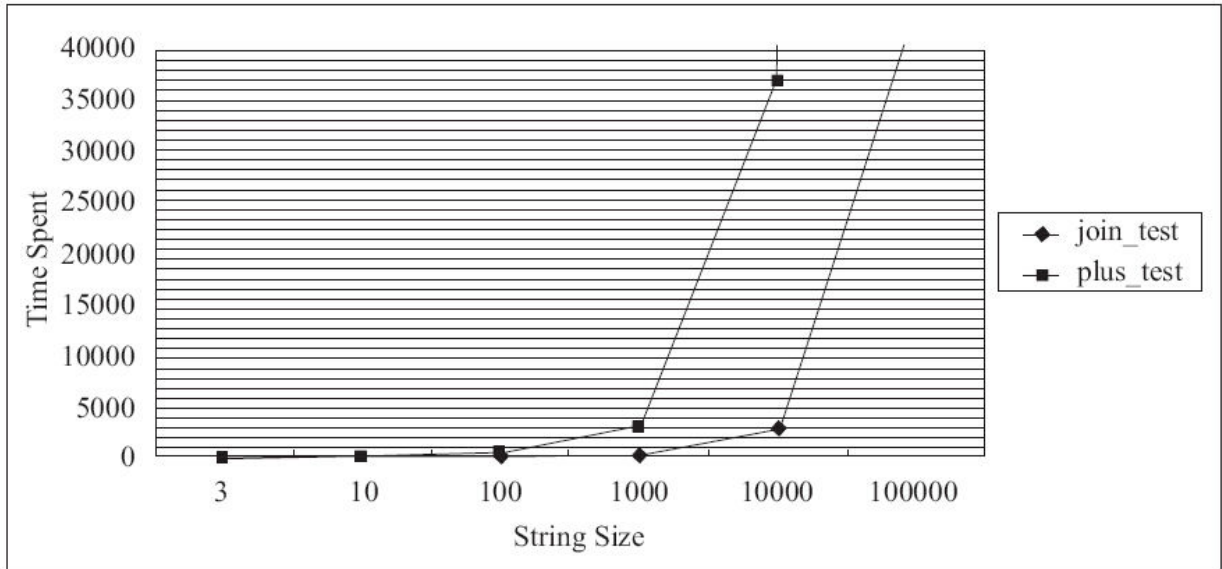


图3-3 join_test()和plus_test()耗时比较图（运行时间扩大10000倍）

从分析测试结果图表我们不难发现：分别使用join()方法和使用+操作符来连接字符串，join()方法的效率要高于+操作符，特别是字符串规模较大的时候，join()方法的优势更为明显（如连接数为100000的时候，两者耗时相差上百倍）。造成这种差别的原因在哪里呢？我们来探讨一下。当用操作符+连接字符串的时候，由于字符串是不可变对象，其工作原理实际上是这样的：如果要连接如下字符串：
 $S_1 + S_2 + S_3 + \dots + S_N$ ，执行一次+操作便会在内存中申请一块新的内存空间，并将上一次操作的结果和本次操作的右操作数复制到新申请的内存空间，即当执行 $S_1 + S_2$ 的时候会申请一块内存，并将 S_1 、 S_2 复制到该内存中，依次类推，如图3-4所示。因此，在N个字符串连接的过程中，会产生N-1个中间结果，每产生一个中间结果都需要申请和复制一次内存，总共需要申请N-1次内存，从而严重影响了执行效率。N越大，对内存的申请和复制的次数越多，+操作符的效率就越低。因此，整个字符串连接的过程中，相当于 S_1 被复制N-1次， S_2 被复制N-2次... S_N 复制1次（并不完全等同于 S_1 复制N-1次，因为后续复制都是对中间结果的复制），所以字符串的连接时间复杂度近似为 $O(n^2)$ 。

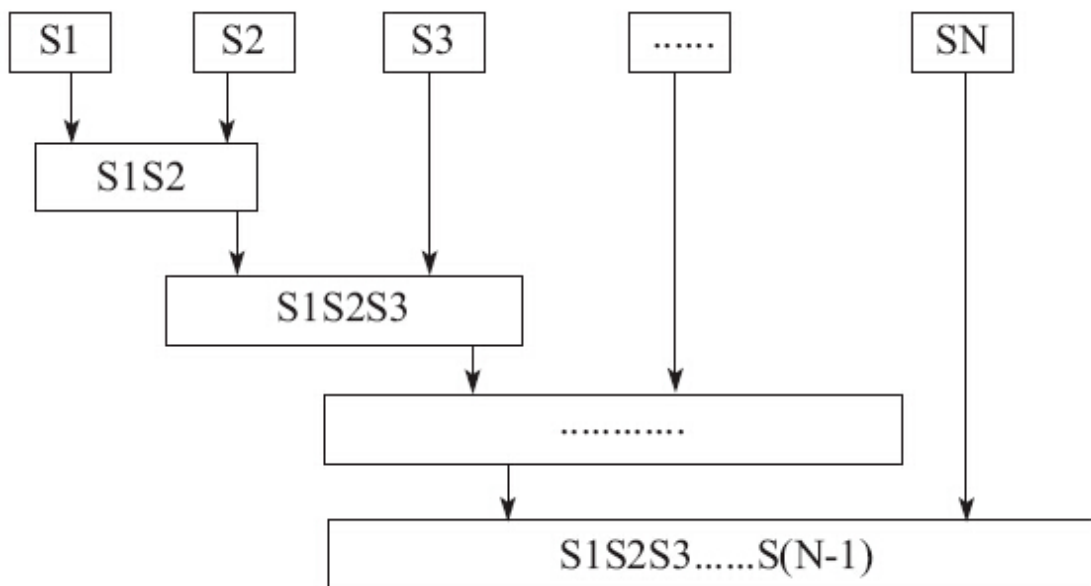


图3-4 操作符+连接字符串示意图

而当用`join()`方法连接字符串的时候，会首先计算需要申请的总的内存空间，然后一次性申请所需内存并将字符序列中的每一个元素复制到内存中去，所以`join`操作的时间复杂度为 $O(n)$ 。

因此，字符串的连接，特别是大规模字符串的处理，应该尽量优先使用`join`而不是`+`。

建议28： 格式化字符串时尽量使用.format方式而不是%

Python中内置的%操作符和.format方式都可用于格式化字符串。先来看看这两种具体格式化方法的基本语法形式和常见用法。

%操作符根据转换说明符所规定的格式返回一串格式化后的字符串，转换说明符的基本形式为：% [转换标记][宽度 [. 精确度]]转换类型。其中常见的转换标记和转换类型分别如表3-2和表3-3所示。如果未指定宽度，则默认输出为字符串本身的宽度。

表3-2 格式化字符串转换标记

转换标记	解 释
-	表示左对齐
+	在正数之前加上 +
(a space)	表示正数之前保留空格
#	在八进制数前面显示零 ('0')，在十六进制前面显示 '0x' 或者 '0X'
0	表示转换值若位数不够则用 0 填充而非默认的空格

表3-3 格式化字符串转换类型

转换类型	解 释
c	转换为单个字符，对于数字将转换该值所对应的 ASCII 码
s	转换为字符串，对于非字符串对象，将默认调用 str() 函数进行转换
r	用 repr() 函数进行字符串转换
i d	转换为带符号的十进制数
u	转换为不带符号的十进制数
o	转换为不带符号的八进制
x X	转换为不带符号的十六进制
e E	表示为科学计数法表示的浮点数
f F	转成浮点数（小数部分自然截断）
g G	如果指数大于 -4 或者小于精度值则和 e 相同，其他情况与 f 相同；如果指数大于 -4 或者小于精度值则和 E 相同，其他情况与 F 相同

%操作符格式化字符串时有如下几种常见用法:

1) 直接格式化字符或者数值。

```
>>> print "your score is %06.1f" % 9.5
your score is 0009.5
>>>
```

2) 以元组的形式格式化。

```
>>> import math
>>> itemname = 'circumference'
>>> radius = 3
>>> print "the %s of a circle with radius %f is %0.3f " %(itemname,radius,math.p
i*radius*2)
the circumference of a circle with radius 3.000000 is 18.850
>>>
```

3) 以字典的形式格式化。

```
>>> itemdict = {'itemname':'circumference','radius':3,'value':math.pi*radius*2}
>>> print "the %(itemname)s of a circle with radius %(radius)f is  %(value)0.3f
" % itemdict
the circumference of a circle with radius 3.000000 is  18.850
>>>
```

.format方式格式化字符串的基本语法为: [[填充符]对齐方式][符号][#][0][宽度][,][.精确度][转换类型]。其中填充符可以是除了“{”和“}”符号之外的任意符号, 对齐方式和符号分别如表3-4和表3-5所示。转换类型跟%操作符的转换类型类似, 可以参见表3-3。

表3-4 .format方式格式化字符串的对齐方式

对齐方式	解 释
<	表示左对齐, 是大多数对象为默认的对齐方式
>	表示右对齐, 数值默认的对齐方式
=	仅对数值类型有效, 如果有符号的话, 在符号后数值前进行填充, 如 -000029
^	居中对齐, 用空格进行填充

表3-5 .format方式格式化字符串符号列表

符号	解 释
+	正数前加 +，负数前加 -
-	正数前不加符号，负数前加 -，为数值的默认形式
空格	正数前加空格，负数前加 -

.format方法几种常见的用法如下：

1) 使用位置符号。

```
>>> "The number {0:}, in hex is: {0:#x}, the number {1} in oct is {1:#o}".format(
4746, 45)
'The number 4,746 in hex is: 0x128a, the number 45 in oct is 0o55'
>>>
#
其中{0}
表示format
方法中对应的第一个参数，{1}
表示format
方法对应的第二个参数，依次递推
```

2) 使用名称。

```
>>> print "the max number is {max}, the min number is {min}, the average number is
{average:0.3f}".format(max=189, min=12.6, average=23.5)
the max number is 189, the min number is 12.6, the average number is 23.500
```

3) 通过属性。

```
>>> class Customer(object):
...     def __init__(self, name, gender, phone):
...         self.name = name
...         self.gender = gender
...         self.phone = phone
...     def __str__(self):
...         return 'Customer({self.name},{self.gender},{self.phone})'.format
(self=self)
#
通过str()
函数返回格式化的结果
...
>>> str(Customer("Lisa", "Female", "67889"))
'Customer(Lisa, Female, 67889)'
>>>
```

4) 格式化元组的具体项。

```
>>> point = (1,3)
>>> 'X:{0[0]};Y:{0[1]}'.format(point)
'X:1;Y:3'
>>>
```

在了解了两种字符串格式的基本用法之后我们再回到本节开头提出的问题：为什么要尽量使用**format**方式而不是**%**操作符来格式化字符串。

理由一： **format**方式在使用上较**%**操作符更为灵活。使用**format**方式时，参数的顺序与格式化的顺序不必完全相同。如：

```
>>> "The number {1} in hex is: {1:#x},the number {0} in oct is {0:#o}".format(4746,45)
'The number 45 in hex is: 0x2d,the number 4746 in oct is 0o11212'
```

上例中格式化的顺序为{1}，{0}，其对应的参数声明的顺序却相反，{1}与45对应，而用**%**方法需要使用字典形式才能达到同样的目的。

理由二： **format**方式可以方便地作为参数传递。

```
>>> weather=[("Monday","rain"),("Tuesday","sunny"),("Wednesday","sunny"),("Thursday","rain"),("Friday","cloudy")]
>>> formatter = "Weather of '{0[0]}' is '{0[1]}'.format
>>> for item in map(formatter,weather):#format
方法作为第一个参数传递给map
函数
...     print item
...
Weather of 'Monday' is 'rain'
Weather of 'Tuesday' is 'sunny'
Weather of 'Wednesday' is 'sunny'
Weather of 'Thursday' is 'rain'
Weather of 'Friday' is 'cloudy'
>>>
```

理由三： %最终会被.format方式所代替。这个理由可以认为是最直接的原因，根据Python的官方文档

(<http://docs.python.org/2/library/stdtypes.html#string-formatting>)，.format()方法最终会取代%，在Python3.0中.format方法是推荐使用的方法，而之所以仍然保留%操作符是为了保持向后兼容。

理由四： %方法在某些特殊情况下使用时需要特别小心。

```
>>> itemname = ("mouse", "mobilephone", "cup")
>>> print "itemlist are %s" %(itemname)          #
使用%
方法格式化元组
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not all arguments converted during string formatting
>>>
>>> print "itemlist are %s" %(itemname,)          #
注意后面的符号,
itemlist are ('mouse', 'mobilephone', 'cup')
>>> print "itemlist are {}".format(itemname)      #
使用format
方法直接格式化不会抛出异常
itemlist are ('mouse', 'mobilephone', 'cup')
>>>
```

上述例子中本意是把itemname看做一个整体来进行格式化，但直接使用时却抛出TypeError，对于%直接格式化字符的这种形式，如果字符本身为元组，则需要使用在%使用(itemname,)这种形式才能避免错误，注意逗号。

建议29：区别对待可变对象和不可变对象

Python中一切皆对象，每一个对象都有一个唯一的标示符（`id()`）、类型（`type()`）以及值。对象根据其值能否修改分为可变对象和不可变对象，其中数字、字符串、元组属于不可变对象，字典以及列表、字节数组属于可变对象。而“菜鸟”常常会试图修改字符串中某个字符。看下面这个例子：

```
teststr = "I am a pytlon string"
teststr[11]='h'
print teststr
```

字符串为不可变对象，任何对字符串中某个字符的修改都会抛出异常。修改字符串中某个字符可以采用如下方式：

```
>>> teststr = "I am a pytlon string"
>>> import array
>>> a = array.array('c',teststr)
>>> a[10]='h'
>>> a
array('c', 'I am a Python string')
>>> a.tostring()
'I am a Python string'
```

再来看下面一段程序：

```
class Student(object):
    def __init__(self,name,course=[]):
        self.name=name
        self.course=course
    def addcourse(self,coursename):
        self.course.append(coursename)
    def printcourse(self):
        for item in self.course:
            print item
stuA=Student("Wang yi")
stuA.addcourse("English")
stuA.addcourse("Math")
print stuA.name + "'s course:"
stuA.printcourse()
print "-----"
```

```
stuB=Student("Li san")
stuB.addcourse("Chinese")
stuB.addcourse("Physics")
print stuB.name + "'s course:"
stuB.printcourse()
```

上述代码清单描述的是两个不同的学生选择不同课程的场景。这个程序有什么问题吗？通过运行程序得到如下输出结果：

```
Wang yi's course:
English
Math
-----
Li san's course:
English
Math
Chinese
Physics
```

你会诧异地发现Li san同学所选的多了English和Math两门课程。这是怎么回事呢？通过查看`id(stuA.course)`和`id(stuB.course)`，我们发现这两个值是一样的，也就是说在内存中指的是同一块地址，但`stuA`和`stuB`本身却是两个不同的对象。在实例化这两个对象的时候，这两个对象被分配了不同的内存空间，并且调用`init()`函数进行初始化。但由于`init()`函数的第二个参数是个默认参数，默认参数在函数被调用的时候仅仅被评估一次，以后都会使用第一次评估的结果，因此实际上对象空间里面`course`所指向的是`list`的地址，每次操作的实际上是`list`所指向的具体列表。这是我们在将可变对象作为函数默认参数的时候要特别警惕的问题，对可变对象的更改会直接影响原对象。要解决上述例子中的问题，最好的方法是传入`None`作为默认参数，在创建对象的时候动态生成列表。具体代码如下：

```
def __init__(self,name,course=None):
    self.name=name
    if course is None:course=[]
    self.course=course
```

对于可变对象，还有一个问题是需要注意的。我们通过以下例子来说明：

```
>>> list1=['a','b','c']
>>> list2 = list1
>>> list1.append('d')
>>> list1
['a', 'b', 'c', 'd']
>>> list2
①list2
也会发生变化
['a', 'b', 'c', 'd']
>>> list3 = list1[:]
②切片操作相当于浅拷贝
>>> list3.remove('a')
>>> list3
['b', 'c', 'd']
>>> list1
['a', 'b', 'c', 'd']
>>> list2
['a', 'b', 'c', 'd']
>>> id(list3)
③重新指向一块内存
14075304
>>> id(list1)
13418864
>>> id(list2)
13418864
```

上面的例子中对list1的切片操作实际会重新生成一个对象，因为切片操作相当于浅拷贝，因此对list3的操作并不会改变list1和list2本身。我们再来看以下不可变对象的简单例子：

```
a=1
a+=2
print a
```

我们会发现此时a的值变为3，这是理所当然的，可是仔细一想a是属于数值类型，是不可变对象，怎么会发生改变呢？实际上Python中变量a存放的是数值1在内存中的地址，数值1本身才是不可变对象。在上面的过程中所改变的是a所指向的对象的地址，数值1并没有发生改变，当执行a+=2的时候重新分配了一块内存地址存放结果，并将a的引

用改为该内存地址，而对象1所在的内存空间会最终被垃圾回收器回收。上述分析的图示如图3-5所示。

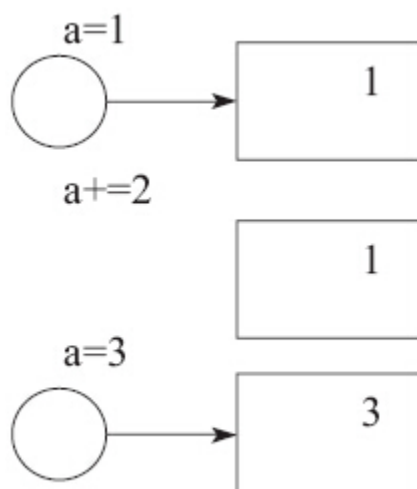


图3-5 a在内存中的变化示意图

通过id()函数分析也可以证实上述变化过程。

```
>>> a=1
>>> id(a)
12184696
>>> id(1)
12184696
①id(a)
和id(1)
的值并不相等
12184696
>>> a+=2
>>> a
3
>>> id(a)
12184672
②id(a)
的值发生改变
12184672
>>> id(3)
12184672
③id(a)
和id(3)
的值相同
12184672
```

id()函数分析也可以证实上述变化过程，对于不可变对象来说，当我们对其进行相关操作的时候，Python实际上仍然保持原来的值而是重新创建一个新的对象，所以字符串对象不允许以索引的方式进行赋

值，当有两个对象同时指向一个字符串对象的时候，对其中一个对象的操作并不会影响另一个对象。

```
>>> str1 = "hello world"
>>> str2 = str1
>>> str1 = str1[:-5]
>>> str1
'hello '
>>> str2
'hello world'
>>>
```

建议30: []、()和{}: 一致的容器初始化形式

列表是一个很有用的数据结构，它在Python中属于可变对象，列表中的元素没有限制，可以重复可以嵌套，操作上支持对单个元素的读取和修改，还支持分片、排序、插入、删除等。由于其灵活性，在实际应用中经常会看到它的身影。下面的程序遍历列表中的每个元素，并根据要求（去掉单词所包含的空格后首字母是否为大写）生成一个新的list。

```
words = [' Are', ' abandon', 'Passion','Business',' fruit ', 'quit']
size = len(words)
newlist = []
for i in range(size):
    if words[i].strip().istitle():
        newlist.append(words[i])
print newlist
```

那么，这段程序有什么问题吗？就程序本身来说并没有什么明显的问题，但有更好的实现方式。这就是列表解析（list comprehension）所涉及的内容了。先来了解一下列表解析的基本知识点。

列表解析的语法为：[**expr** for **iter_item** in **iterable** if **cond_expr**]。它迭代**iterable** 中的每一个元素，当条件满足的时候便根据表达式**expr**计算的内容生成一个元素并放入新的列表中，依次类推，并最终返回整个列表。在语法上，它等价于下面代码段：

```
Newlist = []
for iter_item in iterable:
    if cond_expr:
        Newlist.append(expr)
```

其中条件表达式不是必需的，如果没有条件表达式，就直接将expr中计算出的元素加入List中。列表解析的使用非常灵活。

1) 支持多重嵌套。如果需要生成一个二维列表可以使用列表解析嵌套的方式。示例如下：

```
>>> nested_list = [['Hello', 'World'], ['Goodbye', 'World']]
>>> nested_list = [[s.upper() for s in xs] for xs in nested_list]
>>> print nested_list
[['HELLO', 'WORLD'], ['GOODBYE', 'WORLD']]
>>>
```

2) 支持多重迭代。下面的例子中a、b分别对应两个列表中的元素，[(a,b) for a in ['a','1',1,2] for b in ['1',3,4,'b'] if a != b]表示：列表['a','1',1,2]和['1',3,4,'b']依次求笛卡儿积之后并去掉元素值相等的元组之后所剩下的元组的集合。

```
>>> [(a,b) for a in ['a','1',1,2] for b in ['1',3,4,'b'] if a != b]
[('a', '1'), ('a', 3), ('a', 4), ('a', 'b'), ('1', 3), ('1', 4), ('1', 'b'), (1, '1'), (1, 3), (1, 4), (1, 'b'), (2, '1'), (2, 3), (2, 4), (2, 'b')]
>>>
```

3) 列表解析语法中的表达式可以是简单表达式，也可以是复杂表达式，甚至是函数。

```
>>> def f(v):
...     if v%2 == 0:
...         v = v**2
...     else:
...         v = v+1
...     return v
...
>>> [f(v) for v in [2,3,4,-1] if v>0]          #
表达式可以是函数
[4, 4, 16]
>>> [v**2 if v%2 ==0 else v+1 for v in [2,3,4,-1] if v>0]#
也可以是普通计算
[4, 4, 16]
>>>
```

4) 列表解析语法中的`iterable`可以是任意可迭代对象。下面的例子中把文件句柄当做一个可迭代对象，可轻易读出文件内容。

```
fh = open("test.txt", "r")
result = [i for i in fh if "abc" in i] #
文件句柄可以当做可迭代对象
print result
```

了解完列表解析的基本知识之后，本节开头的例子你应该知道怎么使用更为简洁了。那么，为什么要推荐在需要生成列表的时候使用列表解析呢？

1) 使用列表解析更为直观清晰，代码更为简洁。本节开头的例子改用列表解析，直接可将代码行数减少为2行，这在大型复杂应用程序中尤为重要，因为这意味着潜在的缺陷较小，同时代码清晰直观，更容易阅读和理解，可维护性更强。

2) 列表解析的效率更高。本节开头的例子用两种不同方法实现后进行测试，测试结果表明列表解析在时间上有一定的优势，这主要是因为普通循环生成List的时候一般需要多次调用`append()`函数，增加了额外的时间开销。需要说明的是对于大数据处理，列表解析并不是一个最佳选择，过多的内存消耗可能会导致`MemoryError`。

除了列表可以使用列表解析的语法之外，其他几种内置的数据结构也支持，比如元组（`tuple`）的初始化语法是（`expr for iter_item in iterable if cond_expr`），而集合（`set`）的初始化语法是`{expr for iter_item in iterable if cond_expr}`，甚至字典（`dict`）也有类似的语法`{expr1, expr2 for iter_item in iterable if cond_expr}`。它们的使用类似列表解析，但需要注意，因为元组也适用赋值语句的装箱和拆箱机制，所以需要注意（1）与（1，）是不同的：前者为数字1，后者才代表元组（注意1后面的“，”）。

```
>>> type((1))
<type 'int'>
>>> type((1,))
<type 'tuple'>
>>>
```

此外，当函数接受一个可迭代对象参数时，可以使用元组的简写方式。

```
>>> def foo(a):
...     for i in a:
...         print i
...
>>> foo([1, 2, 3])
1
2
3
>>> foo(i for i in range(3) if i % 2 == 0)
0
2
```

建议31： 记住函数传参既不是传值也不是传引用

Python中的函数参数到底是传值还是传引用呢？这是许多人在学习过程中会纠结的一个问题，很多论坛也有这样的讨论。总结来说基本有3个观点：传引用；传值；可变对象传引用，不可变对象传值。这3个观点到底哪个正确呢？我们逐一讨论。

1) 传引用。先来看一个非常简单的例子（请不要因为例子太简单而不以为然，小故事往往蕴含大道理，它照样能说明问题）。

示例一：

```
>>> def inc(n):
...     print id(n)
...     n = n+1
...     print id(n)
...
>>> n = 3
>>> id(n)
34450040
>>> inc(n)
34450040 #
修改之前的n
的id
值
34450028 #
修改之后的n
的id
值
>>> print n
3
>>>
```

按照传引用的概念，上面的例子期望的输出应该是4，并且inc()函数里面执行操作`n=n+1`的前后n的id值应该是不变的。可是事实是不是这样的呢？非也，从输出结果来看n的值还是不变，但id(n)的值在函数体前后却不一致。显然，传引用这个说法是不恰当的。

2) 传值。来看一个示例。

示例二:

```
>>> def change_list(originator_list):
...     print "originator list is:",originator_list
...     new_list = originator_list
...     new_list.append("I am new")
...     print "new list is:", new_list
...     return new_list
...
>>>
>>> originator_list = ['a','b','c']
>>> new_list = change_list(originator_list)
originator list is: ['a', 'b', 'c']
new list is: ['a', 'b', 'c', 'I am new']
>>> print new_list
['a', 'b', 'c', 'I am new']
>>> print originator_list
['a', 'b', 'c', 'I am new']
>>>
```

传值通俗来讲就是这个意思：你在内存中有一个位置，我也有一个位置，我把我的值复制给你，以后你做什么就跟我没关系了，我是我，你是你，咱俩井水不犯河水。可是上面的程序输出根本就不是这么一回事，显然`change_list()`函数没有遵守约定，调用该函数之后`originator_list`也发生了改变，这明显侵犯了`originator_list`的权利。这么看来传值这个说法也不合适。

3) 可变对象传引用，不可变对象传值。这个说法最靠谱，很多人也是这么理解的，但这个说法到底是否准确呢？再来看一个示例。

示例三:

```
>>> def change_me(org_list):
...     print id(org_list)
...     new_list = org_list
...     print id(new_list)
...     if len(new_list)>5:
...         new_list = ['a','b','c']
...     for i,e in enumerate(new_list):
...         if isinstance(e,list):
...             new_list[i]="***"    #
...
将元素为list
类型的替换为***
```

```
...     print new_list
...     print id(new_list)
...
>>>
```

传入参数`org_list`为列表，属于可变对象，按照可变对象传引用的理解，`new_list`和`org_list`指向同一块内存，因此两者的`id`值输出一致，任何对`new_list`所执行的内容的操作会直接反应到`org_list`，也就是说修改`new_list`会导致`org_list`的直接修改，对吧？来看测试例子。

```
>>> test1 = [1,['a',1,3],[2,1],6]
>>> change_me(test1)                                #test1
的元素个数小于5
35260216
35260216
[1, '***', '***', 6]
                                     #test1
中所有list
类型的元素都替换为了***
35260216
>>> print test1
[1, '***', '***', 6]
>>> test2=[1,2,3,4,5,6,[1]]
                                     #test1
中元素的个数大于5
>>> change_me(test2)
35260136
35260136
['a', 'b', 'c']
35250664 #new_list
的id
值发生了改变
>>> print test2                                #test2
并没有发生改变
[1, 2, 3, 4, 5, 6, [1]]
>>>
```

对于`test1`、`new_list`和`org_list`的表现和我们理解的传引用确实一致，最后`test1`被修改为`[1, '***', '***', 6]`，但对于输入`test2`、`new_list`和`org_list`的`id`输出在进行列表相关的操作前是一致的，但操作之后`new_list`的`id`值却变为35250664，整个`test2`在调用函数`change_me()`后却没有发生任何改变，可是按照传引用的理解期望输出应该是`['a','b','c']`，似乎可变对象传引用这个说法也不恰当。

那么Python函数中参数传递的机制到底是怎样的？要明白这个概念，首先要理解：Python中的赋值与我们所理解的C/C++等语言中的赋值的意思并不一样。如果有如下语句：

```
a =5
, b = a
, b=7 ;
```

我们分别来看一下在C/C++以及Python中是如何赋值的。

如图3-6所示，C/C++中当执行b=a的时候，在内存中申请一块内存并将a的值复制到该内存中；当执行b=7之后是将b对应的值从5修改为7。

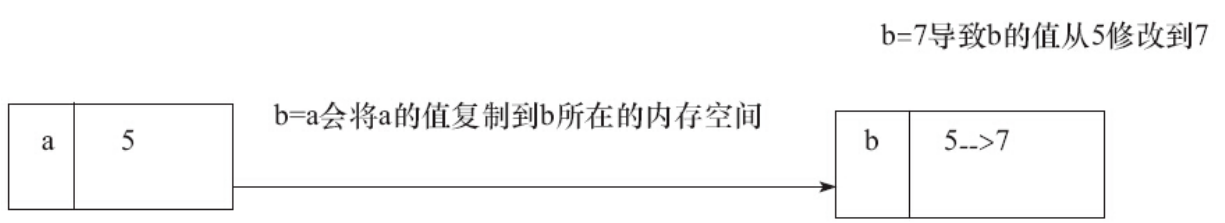


图3-6 C/C++赋值时内存的变化

但在Python中赋值并不是复制，b=a操作使得b与a引用同一个对象。而b=7则是将b指向对象7，如图3-7所示。

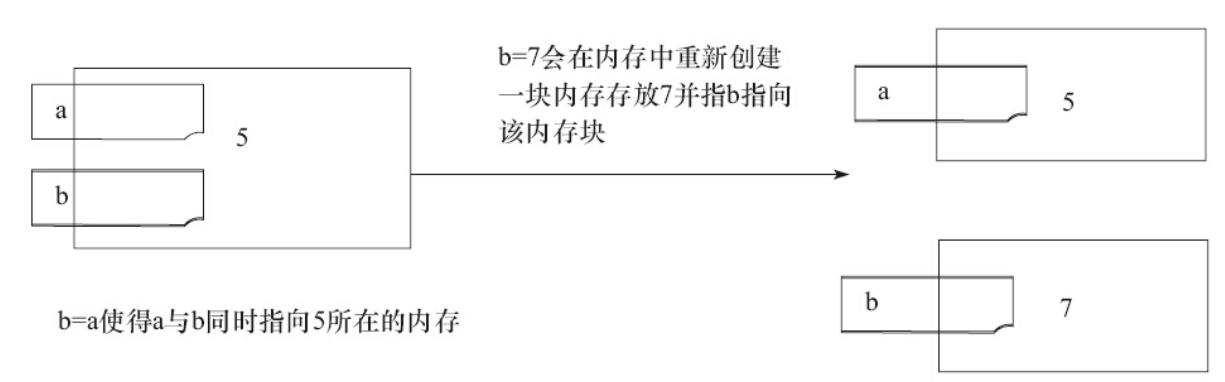


图3-7 Python中赋值语句对应的内存变化

我们通过以下示例来验证上面所述的过程：

```
>>> a = 5
>>> id(a)
34450016
>>> b = a
>>> id(b)          #b=a
之后b
的id()
值和a
一样
34450016
>>> b = 7
>>> id(b)          #b=7
之后b
指向对象7
, id()
值发生改变
34449992
>>> id(a)
34450016
```

从输出可以看出，`b=a`赋值后`b`的`id()`输出和`a`一样，但`b=7`操作后`b`指向另外一块空间。可以简单理解为，`b=a`传递的是对象的引用，其过程类似于贴“标签”，5和7是实实在在的内存空间，执行`a=5`相当于申请一块内存空间代表对象5并在上面贴上标签`a`，这样`a`和5便绑定在一起了。而`b=a`相当于对标签`a`创建了一个别名，因此它们实际都指向5。但`b=7`操作之后标签`b`重新贴到7所代表的对象上去了，而此时5仅有标签`a`。

理解了上述背景，再回头来看看前面的例子就很好理解了。对于示例一，`n=n+1`，由于`n`为数字，是不可变对象，`n+1`会重新申请一块内存，其值为`n+1`，并在函数体中创建局部变量`n`指向它。当调用完函数`inc(n)`之后，函数体中的局部变量在函数外并不可见，此时的`n`代表函数体外的命名空间所对应的`n`，值还是3。而在示例三中，当`org_list`的长度大于5的时候，`new_list = ['a','b','c']`操作重新创建了一块内存并将`new_list`指向它。当传入参数为`test2=[1,2,3,4,5,6,[1]]`的时候，函数的执行并没有改变该列表的值。

因此，对于Python函数参数是传值还是传引用这个问题的答案是：都不是。正确的叫法应该是传对象（**call by object**）或者说传对象的引用（**call-by-object-reference**）。函数参数在传递的过程中将整个对象传入，对可变对象的修改在函数外部以及内部都可见，调用者和被调用者之间共享这个对象，而对于不可变对象，由于并不能真正被修改，因此，修改往往是通过生成一个新对象然后赋值来实现的。

建议32：警惕默认参数潜在的问题

默认参数可以给函数的使用带来很大的灵活性，当函数调用没有指定与形参对应的实参时就会自动使用默认参数。

```
>>> def appendtest(newitem, lista = []):          #
默认参数为空列表
...     print id(lista)
...     lista.append(newitem)
...     print id(lista)
...     return lista
...
>>>
>>> appendtest('a', ['b', 2, 4, [1, 2]])
39644216
39644216
['b', 2, 4, [1, 2], 'a']
>>>
```

现在请读者思考这么一个问题：如果第二个参数采取默认参数，连续调用两次`appendtest(1)`、`appendtest('a')`，函数的返回值是多少？期望的结果应该是`[1]`和`['a']`，对吧？可是实际情况却输出了`[1]`和`[1, 'a']`。那么这是为什么呢？

`def`在Python中是一个可执行的语句，当解释器执行`def`的时候，默认参数也会被计算，并存在函数的`func_defaults`属性中。由于Python中函数参数传递的是对象，可变对象在调用者和被调用者之间共享，因此当首次调用`appendtest(1)`的时候，`[]`变为`[1]`，而再次调用的时候由于默认参数不会重新计算，在`[1]`的基础上便变为了`[1, 'a']`。我们可以通过查看函数的`func_defaults`来确认这一点。

```
>>> appendtest(1)
39022960
39022960
[1]
>>> appendtest.func_defaults          #
第一次调用后默认参数变为[1]
([1],)
```

```
>>> appendtest('a')
39022960
39022960
[1, 'a']
>>> appendtest.func_defaults
#
经过第二次调用
变为[1,
'a'
']
([1, 'a'],)
>>>
>>> appendtest.func_defaults[0][:] = []
#
可以直接修改func_defaults
属性
>>> appendtest.func_defaults
([],)
```

如果不想让默认参数所指向的对象在所有的函数调用中被共享，而是在函数调用的过程中动态生成，可以在定义的时候使用**None**对象作为占位符。因此本节开头的例子应该修正为如下形式：

```
>>> def appendtest(newitem, lista = None):#
默认参数改为None
...     if lista is None:
...         lista = []
...     lista.append(newitem)
...     return lista
...
```

最后以一个问题结束本节：假设**report()**函数需要传入当前系统的时间并做一些处理，下面两种参数传递方式哪种正确呢？读者应该知道答案了！

```
>>> import time
>>> def report(when = time.time()):
...     pass
...
>>> def report(when = time.time):
...     pass
...
```

建议33：慎用变长参数

Python支持可变长度的参数列表，可以通过在函数定义的时候使用`*args`和`**kwargs`这两个特殊语法来实现（`args`和`kwargs`可以替换成任意你喜欢的变量名）。先来看两个可变长参数使用的例子。

1) 使用`*args`来实现可变参数列表：`*args`用于接受一个包装为元组形式的参数列表来传递非关键字参数，参数个数可以任意，如下例所示。

```
def SumFun(*args):
    result = 0
    for x in args[0:]:
        result += x
    return result
print SumFun(2,4)
print SumFun(1,2,3,4,5)
print SumFun()
```

2) 使用`**kwargs`接受字典形式的关键字参数列表，其中字典的键值对分别表示不可变参数的参数名和值。如下例中`apple`表示参数名，而`fruit`为其对应的`value`，可以是一个或者多个键值对。

```
def category_table(**kwargs):
    for name, value in kwargs.items():
        print '{0} is a kind of {1}'.format(name, value)
category_table(apple = 'fruit', carrot = 'vegetable', Python = 'programming language')
category_table(BMW = 'Car')
```

如果一个函数中同时定义了普通参数、默认参数，以及上述两种形式的可变参数，那么使用情况又是怎样的呢？来看一个简单的问题。

```
def set_axis(x,y, xlabel="x", ylabel="y", *args, **kwargs):  
    pass
```

对于上面的函数下面几种调用方式哪些是不正确的呢？

```
set_axis(2,3, "test1", "test2","test3", my_kwarg="test3")  
①  
set_axis(2,3, my_kwarg="test3")  
set_axis(2,3, "test1",my_kwarg="test3")  
②  
set_axis("test1", "test2", xlabel="new_x",ylabel = "new_y", my_kwarg="test3")  
set_axis(2,"test1", "test2",ylabel = "new_y", my_kwarg="test3")
```

答案是：上面的所有调用方式都是合法的！实际上在4种不同形式的参数同时存在的情况下，会首先满足普通参数，然后是默认参数。如果剩余的参数个数能够覆盖所有的默认参数，则默认参数会使用传递时候的值，如标注①处的函数在调用的时候xlabel和ylabel的值分别为“test1”和“test2”；如果剩余参数个数不够，则尽最大可能满足默认参数的值，标注②中xlabel值为“test1”，而ylabel则使用默认参数y。除此之外其余的参数除了键值对以外所有的参数都将作为args的可变参数，kwargs则与键值对对应。那么，为什么要慎用可变长度参数呢？原因如下：

1) 使用过于灵活。上面的示例set_axis()随随便便就能写出好几种不同形式的调用方式。在混合普通参数或者默认参数的情况下，变长参数意味着这个函数的签名不够清晰，存在多种调用方式。如果调用者需要花费过多的时间来研究你的方法该如何调用，显然这并不是值得提倡的方式，即使你认为它很炫，很高端大气上档次，但在团队合作开发的项目中，千万不要有这种想法，团队开发不是你的个人竞技场，代码编写从某种程度上说也是一种沟通，清晰准确是一个很重要的指标。另外变长参数可能会破坏程序的健壮性。

2) 如果一个函数的参数列表很长，虽然可以通过使用*args和**kwargs来简化函数的定义，但通常这意味着这个函数可以有更好的实现方式，应该被重构。前面的例子中SumFun(*args)如果改为def Sum(seq)，在函数调用之前将需要传递的参数保存在一个序列中，如seq = (1,2,3,4,5,6,7)，再将序列seq作为参数传入Sum中也是一个不错的选择。同样如果使用**kwargs的目的仅仅是为了传入一个字典，这将是一个非常糟糕的选择。

3) 可变长参数适合在下列情况下使用（不仅限于以下场景）：

·为函数添加一个装饰器。

```
>>> def mydecorator(fun):
...     def new(*args,**kwargs):
...         # ...
...         return fun(*args,**kwargs)
...     return new
...
>>>
```

·如果参数的数目不确定，可以考虑使用变长参数。如配置文件test.cfg内容如下：

```
[Defaults]
name = test
version = 1.0
platform = windows
func(**kwargs)
用于读取一些配置文件中的值并进行全局变量初始化。
from ConfigParser import ConfigParser
conf = ConfigParser()
conf.read('test.cfg')
conf_dict = dict(conf.items('Defaults'))
def func(**kwargs):
    kwargs.update(conf_dict)
    global name
    name = kwargs.get('name')
    global version
    version = kwargs.get('version')
    global platform
    platform = kwargs.get('platform')
```

·用来实现函数的多态或者在继承情况下子类需要调用父类的某些方法的时候。

```
>>> class A(object):
...     def somefun(self,p1,p2):
...         pass
...
>>> class B(A):
...     def myfun(self,p3,*args,**kwargs):
...         super(B,self).somefun(*args,**kwargs)
...
>>>
```

建议34： 深入理解str()和repr()的区别

函数str()和repr()都可以将Python中的对象转换为字符串，它们的使用以及输出都非常相似，以至于很多人认为这两者之间并没有区别，但事实是不是这样呢？先来看对于不同类型的输入，这两个函数的输出有何异同，如表3-6所示。

表3-6 str()和repr()函数在不同情况下输出比较

输入	str()	repr()
a = 1	'1'	'1'
a = 0.1	'0.1'	'0.1'
a = "abc"	'abc'	'''abc'''
a = (1,2,3)	'(1, 2, 3)'	'(1, 2, 3)'

(续)

输入	str()	repr()
class A(object): pass	"<class ' __main__ .A'>"	"<class ' __main__ .A'>"
a = A()	'< __main__ .A object at 0x0228C6B0>'	'< __main__ .A object at 0x0228C6B0>'
def f(): pass	'<function f at 0x02284830>'	'<function f at 0x02284830>'
a = {'a':1}	"{'a': 1}"	"{'a': 1}"
a = 2/3.0 ①	'0.6666666666666667'	'0.6666666666666666'
a = [1,2,3]	'[1, 2, 3]'	'[1, 2, 3]'
a = Decimal(1.25) ②	'1.25'	"Decimal('1.25')"
a = date.today() ③	'2013-07-20'	'datetime.date(2013, 7, 20)'
a = "\r\n" ④	"\r\n"	""\r\\n""

从表3-6看出，repr()和str()对于大多数数据类型的输出基本一致，因此混淆也就不难理解了。但它们也存在不一致的情况。那么，这两者之间到底有什么区别呢？总结来说有以下几点：

1) 两者之间的目标不同: `str()`主要面向用户, 其目的是可读性, 返回形式为用户友好性和可读性都较强的字符串类型; 而`repr()`面向的是Python解释器, 或者说开发人员, 其目的是准确性, 其返回值表示Python解释器内部的含义, 常作为编程人员debug用途。

2) 在解释器中直接输入a时默认调用`repr()`函数, 而`print a`则调用`str()`函数。

3) `repr()`的返回值一般可以用`eval()`函数来还原对象, 通常来说有如下等式。

```
obj == eval(repr(obj))
```

但需要提醒的是, 这个等式不是所有情况下都成立, 如用户重新实现的`repr()`方法如下。

```
>>> s="' '"
>>> str(s)
"' '"
>>> repr(s)
'"\ \'"
>>> eval(repr(s)) == s
True
>>> eval(str(s))
' '
>>> eval(str(s)) == s
False
>>>
```

4) 这两个方法分别调用内建的`__str__()`和`__repr__()`方法, 一般来说在类中都应该定义`__repr__()`方法, 而`__str__()`方法则为可选, 当可读性比准确性更为重要的时候应该考虑定义`__str__()`方法。如果类中没有定义`__str__()`方法, 则默认会使用`__repr__()`方法的结果来返回对象的字符串表示形式。用户实现`__repr__()`方法的时候最好保证其返回值可以用`eval()`方法使对象重新还原。

建议35：分清staticmethod和classmethod的适用场景

Python中的静态方法（`staticmethod`）和类方法（`classmethod`）都依赖于装饰器（`decorator`）来实现。其中静态方法的用法如下：

```
class C(object):
    @staticmethod
    def f(arg1, arg2, ...):
```

而类方法的用法如下：

```
class C(object):
    @classmethod
    def f(cls, arg1, arg2, ...):
```

静态方法和类方法都可以通过类名.方法名（如`C.f()`）或者实例.方法名（`C().f()`）的形式来访问。其中静态方法没有常规方法的特殊行为，如绑定、非绑定、隐式参数等规则，而类方法的调用使用类本身作为其隐含参数，但调用本身并不需要显示提供该参数。

```
class A(object):
    def instance_method(self, x):
        print "calling instance method instance_method(%s,%s)"%(self, x)
    @classmethod
    def class_method(cls, x):
        print "calling class_method(%s,%s)"%(cls, x)
    @staticmethod
    def static_method(x):
        print "calling static_method(%s)"%x
a = A()
a.instance_method("test")
#
输出calling instance method instance_method(<__main__.A object at
0x00D66B50>,test)
a.class_method("test")
#
输出calling class_method(<class '__main__.A'>,test)
a.static_method("test")
```

```
#  
输出calling static_method(test)
```

上面的例子是类方法和静态方法的简单应用，从程序的输出可以看出虽然类方法在调用的时候没有显式声明cls，但实际上类本身是作为隐含参数传入的。

在了解完静态方法和类方法的基本知识之后再来研究这样一个问题：为什么需要静态方法和类方法，它们和普通的实例方法之间有什么区别？我们通过对具体问题的研究来回答这些问题。假设有水果类Fruit，它用属性total表示总量，Fruit中已经有方法set()来设置总量，print_total()方法来打印水果数量。类Apple和类Orange继承自Fruit。我们需要分别跟踪不同类型的水果的总量。有好几种方法可以实现这个功能。

方法一： 利用普通的实例方法来实现。

在Apple和Orange类中分别定义类变量total，然后再覆盖基类的set()和print_total()方法，但这会导致代码冗余，因为本质上这些方法所实现的功能相同（读者可以自行完成）。

方法二： 使用类方法实现，具体实现代码清单如下。

```
class Fruit(object):  
    total = 0  
    @classmethod  
    def print_total(cls):  
        print cls.total  
        #print id(Fruit.total)  
        #print id(cls.total)  
    @classmethod  
    def set(cls, value):  
        #print "calling class_method(%s,%s)"%(cls,value)  
        cls.total = value  
class Apple(Fruit):  
    pass  
class Orange(Fruit):  
    Pass  
app1 = Apple()  
app1.set(200)  
app2 = Apple()
```

```
org1 = Orange()
org1.set(300)
org2 = Orange()
app1.print_total() #output 200
org1.print_total() #output 300
```

删除上面代码中的注释语句后运行程序会得到以下结果：

```
calling class_method(<class '__main__.Apple'>, 200)
calling class_method(<class '__main__.Orange'>, 300)
200
12184820----->Fruit
类的类变量
12186364 ----->
动态生成的Apple
类的类变量
300
12184820----->Fruit
类的类变量
13810996 ----->
动态生成的Orange
类的类变量
```

简单分析可知，针对不同种类的水果对象调用set()方法的时候隐形传入的参数为该对象所对应的类，在调用set()的过程中动态生成了对应的类的类变量。这就是classmethod的妙处。请读者自行思考：此处将类方法改为静态方法是否可行呢？

我们再来看一个必须使用类方法而不是静态方法的例子：假设对于每一个Fruit类我们提供3个实例属性：area表示区域代码，category表示种类代码，batch表示批次号。现需要一个方法能够将以area-category-batch形式表示的字符串形式的输入转化为对应的属性并以对象返回。

假设Fruit中有如下初始化方法，并且有静态方法Init_Product()能够满足上面所提的要求。

```
def __init__(self, area="", category="", batch=""):
    self.area = area
    self.category = category
    self.batch = batch
    @staticmethod
```

```
def Init_Product(product_info):
    area, category, batch = map(int, product_info.split('-'))
    fruit = Fruit(area, category, batch)
    return fruit
```

我们首先来看看使用静态方法所带来的问题。

```
app1 = Apple (2,5,10)
org1 = Orange.Init_Product("3-3-9")
print "app1 is instance of Apple:"+str(isinstance(app1, Apple))
print "org1 is instance of Orange:"+str(isinstance(org1, Orange))
```

运行程序我们会发现`isinstance(org1, Orange)`的值为`False`。这不奇怪，因为静态方法实际相当于一个定义在类里面的函数，`Init_Product`返回的实际是`Fruit`的对象，所以它不会是`Orange`的实例。

`Init_Product()`的功能类似于工厂方法，能够根据不同的类型返回对应的类的实例，因此使用静态方法并不能获得期望的结果，类方法才是正确的解决方案。可以针对代码做出如下修改：

```
@classmethod
def Init_Product(cls,product_info):
    area, category, batch = map(int, product_info.split('-'))
    fruit = cls(area, category, batch)
    return fruit
```

也许读者会问：既然这样，静态方法到底有什么用呢？什么情况下可以使用静态方法？继续上面的例子，假设我们还需要一个方法来验证输入的合法性，方法的具体实现如下：

```
def is_input_valid(product_info):
    area, category, batch = map(int, product_info.split('-'))
    try:
        assert 0 <= area <= 10
        assert 0 <= category <= 15
        assert 0 <= batch <= 99
    except AssertionError:
        return False
    return True
```

那么应该将其声明为静态方法还是类方法呢？答案是两者都可，甚至将其作为一个定义在类的外部的函数都是可以的。但仔细分析该方法会发现它既不跟特定的实例相关也不跟特定的类相关，因此将其定义为静态方法是个不错的选择，这样代码能够一目了然。也许你会问：为什么不将该方法定义成外部函数呢？这是因为静态方法定义在类中，较之外部函数，能够更加有效地将代码组织起来，从而使相关代码的垂直距离更近，提高代码的可维护性。当然，如果有一组独立的方法，将其定义在一个模块中，通过模块来访问这些方法也是一个不错的选择。

第4章 库

Python具有丰富的库，掌握所有的库的用法和使用基本上是不可能的，更好的方法是掌握一些常见库的使用和注意事项，对于使用较少的库可以在具体应用时参考其文档以及使用范例。本章主要讨论一些常用库的使用和技巧。

建议36：掌握字符串的基本用法

无名氏说：编程有两件事，一件是处理数值，另一件是处理字符串。要我说，对于商业应用编程来说，处理字符串的代码可能超过八成，所以掌握字符串的基本用法尤其重要。通过Python教程，读者已经掌握了基本的字符串字面量语法，比如u、r前缀等，但对于怎么更好地编写多行的字符串字面量，仍然有个小技巧值得向大家推介。

```
>>> s = ('SELECT * '
...     'FROM atable '
...     'WHERE afield="value"')
>>> s
'SELECT * FROM atable WHERE afield="value"'
```

这就是利用Python遇到未闭合的小括号时会自动将多行代码拼接为一行和把相邻的两个字符串字面量拼接在一起的特性做到的。相比使用3个连续的单（双）引号，这种方式不会把换行符和前导空格也当作字符串的一部分，则更加符合用户的思维习惯。

除了这个小技巧，也许你已经听说过Python中的字符串其实有str和unicode两种。是的，的确如此，虽然在Python 3中已经简化为一种，但如果你还在编写运行在Python 2上的程序，当需要判断变量是否为字符串时，需要注意了。判断一个变量s是不是字符串应使用isinstance(s,basestring)，注意这里的参数是basestring而不是str。

```
>>> a = "hi"
>>> isinstance(a,str)           ... ..
①
True
>>> b =u"Hi"
>>> isinstance(b,str)           ... ..
②
False
>>> isinstance(b,basestring)
True
>>> isinstance(b,unicode)
```

```
True
>>> isinstance(a,unicode)          ... ..
③
False
>>>
```

如标注①所示：`isinstance(a,str)`用于判断一个字符串是不是普通字符串，也就是说其类型是否为`str`；因此当被判断的字符串为`Unicode`的时候，返回`False`，如标注②所示。同样，标注③中`isinstance(a,unicode)`用来判断一个字符串是不是`Unicode`。因此要正确判断一个变量是不是字符串，应该使用`isinstance(s,basestring)`，因为`basestring`才是是`str`和`unicode`的基类，包含了普通字符串和`unicode`类型。

接下来正式开始学习字符串的基本用法。与其他书籍、手册不同，我们将通过性质判定、查找替换、分切与连接、变形、填空与删减等5个方面来学习。首先是性质判定，`str`对象有以下几个方法：

`isalnum()`、`isalpha()`、`isdigit()`、`islower()`、`isupper()`、`isspace()`、`istitle()`、`startswith(prefix[, start[, end]])`、`endswith(suffix[,start[, end]])`，前面几个`is*()`形式的函数很简单，顾名思义无非是判定是否数字、字母、大小写、空白符之类的，`istitle()`作为东方人用得少些，它是判定字符串是否每个单词都有且只有第一个字母是大写的。

```
>>> assert 'Hello World!'.istitle() == True
>>> assert 'Hello World!'.istitle() == False
```

相对于`is*()`这些“小儿科”来说，需要注意的是`*with()`函数族可以接受可选的`start`、`end`参数，善加利用，可以优化性能。另外，自Python 2.5版本起，`*with()`函数族的`prefix`参数可以接受`tuple`类型的实参，当实参中的某个元素能够匹配时，即返回`True`。

接下来是查找与替换，`count(sub[, start[, end]])`、`find(sub[, start[, end]])`、`index(sub[, start[, end]])`、`rfind(sub[, start[,end]])`、`rindex(sub[,`

`start[, end]]`)这些方法都接受`start`、`end`参数，善加利用，可以优化性能。其中`count()`能够查找子串`sub`在字符串中出现的次数，这个数值在调用`replace`方法的时候用得着。此外，需要注意`find()`和`index()`方法的不同：`find()`函数族找不到时返回-1，`index()`函数族则抛出`ValueError`异常。但对于判定是否包含子串的判定并不推荐调用这些方法，而是推荐使用`in`和`not in`操作符。

```
>>> str = "Test if a string contains some special substrings"
>>> if str.find("some") != -1: #
使用find
方法进行判断
...     print "Yes,it contains"
...
Yes,it contains
>>> if "some" in str: #
使用in
方法也可以判断
...     print "Yes,it contains using in"
...
Yes,it contains using in
```

`replace(old, new[,count])`用以替换字符串的某些子串，如果指定`count`参数的话，就最多替换`count`次，如果不指定，就全部替换（跟其他语言不太一样，要注意了）。

然后要掌握字符串的分切与连接，关于连接，会有一节专门进行讲述，在这里，专讲分切。`partition(sep)`、`rpartition(sep)`、`splitlines([keepends])`、`split([sep [,maxsplit]])`、`rsplit([sep[,maxsplit]])`，别看这些方法好像很多，其实只要弄清楚`partition()`和`split()`就可以了。
*`partition()`函数族是2.5版本新增的方法，它接受一个字符串参数，并返回一个3个元素的元组对象。如果`sep`没出现在母串中，返回值是`(sep, "", "")`；否则，返回值的第一个元素是`sep`左端的部分，第二个元素是`sep`自身，第三个元素是`sep`右端的部分。而`split()`的参数`maxsplit`是分切的次数，即最大的分切次数，所以返回值最多有`maxsplit+1`个元素。但`split()`有不少小陷阱，需要注意，比如对于字符串`s`、`s.split()`和`s.split("")`的返回值是不相同的。

```
>>> ' hello world!'.split()
['hello', 'world!']
>>> ' hello world!'.split(' ')
['', '', 'hello', '', '', 'world!']
```

产生差异的原因在于：当忽略sep参数或sep参数为None时与明确给sep赋予字符串值时，split()采用两种不同的算法。对于前者，split()先去除字符串两端的空白符，然后以任意长度的空白字符串作为界定符分切字符串（即连续的空白字符串被当作单一的空白符看待）；对于后者则认为两个连续的sep之间存在一个空字符串。因此对于空字符串（或空白字符串），它们的返回值也是不同的。

```
>>> ''.split()
[]
>>> ''.split(' ')
['']
```

掌握了split()，可以说字符串最大的陷阱已经跨过去了。下面是关于变形的内容。lower()、upper()、capitalize()、swapcase()、title()这些无非是大小写切换的小事，不过需要注意的是title()的功能是将每一个单词的首字母大写，并将单词中的非首字母转换为小写（英文文章的标题通常是这种格式）。

```
>>> 'hello wORLD!'.title()
'Hello World!'
```

因为title()函数并不去除字符串两端的空白符也不会把连续的空白符替换为一个空格，所以不能把title()理解先以空白符分切字符串，然后调用capitalize()处理每个字词以使其首字母大写，再用空格将它们连接在一起。如果你有这样的需求，建议使用string模块中的capwords(s)函数，它能够去除两端的空白符，再将连续的空白符用一个空格代替。

```
>>> ' hello  world!'.title()
' Hello  World!'
>>> string.capwords(' hello  world!')
'Hello World!'
```

看，它们的结果是不相同的！最后，是删减与填充。删减在文本处理是很常用，我们常常得把字符串掐头去尾，就用得上它们。如果 `strip([chars])`、`lstrip([chars])`、`rstrip([chars])` 中的 `chars` 参数没有指定，就是删除空白符，空白符由 `string.whitespace` 常量定义。填充则常用于字符串的输出，借助它们能够排出漂亮的版面。`center(width[, fillchar])`、`ljust(width[, fillchar])`、`rjust(width[, fillchar])`、`zfill(width)`、`expandtabs([tabsize])`，看，有了它们，居中、左对齐、右对齐什么的完全不在话下，这些方法中的 `fillchar` 参数是指用以填充的字符，默认是空格。而 `zfill()` 中的 `z` 是指 `zero`，所以顾名思义，`zfill()` 即是以字符 `0` 进行填充，在输出数值时比较常用。`expandtabs()` 的 `tabsize` 参数默认为 `8`，它的功能是把字符串中的制表符（`tab`）转换为适当数量的空格。

建议37： 按需选择sort()或者sorted()

各种排序算法以及它们的时间复杂度分析是很多企业面试人员在面试时候经常会问到的问题，这也不难理解，在实际的应用过程中确实会遇到各种需要排序的情况，如按照字母表输出一个序列、对记录的多个字段排序等。还好，Python中的排序相对简单，常用的函数有sort()和sorted()两种。这两种函数并不完全相同，各有各的用武之地。我们来具体分析一下。

1) 相比于sort()，sorted()使用的范围更为广泛，两者的函数形式分别如下：

```
sorted(iterable[, cmp[, key[, reverse]])  
s.sort([cmp[, key[, reverse]])
```

这两个方法有以下3个共同的参数：

- cmp**为用户定义的任何比较函数，函数的参数为两个可比较的元素（来自iterable或者list），函数根据第一个参数与第二个参数的关系依次返回-1、0或者+1（第一个参数小于第二个参数则返回负数）。该参数默认值为None。

- key**是带一个参数的函数，用来为每个元素提取比较值，默认为None（即直接比较每个元素）。

- reverse**表示排序结果是否反转。

```
>>> persons = [{'name': 'Jon', 'age': 32}, {'name': 'Alan', 'age': 50}, {'name':  
'Bob', 'age': 23}]  
>>> sorted(persons, key=lambda x: (x['name'], -x['age']))
```

```
[{'age': 50, 'name': 'Alan'}, {'age': 23, 'name': 'Bob'}, {'age': 32, 'name': 'Jon'}]
```

从函数的定义形式可以看出，`sorted()`作用于任意可迭代的对象，而`sort()`一般作用于列表。因此下面的例子中针对元组使用`sort()`方法会抛出`AttributeError`，而使用`sorted()`函数则没有这个问题。

```
>>> a = (1,2,4,2,3)
>>> a.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'sort'
>>> sorted(a)
[1, 2, 2, 3, 4]
```

2) 当排序对象为列表的时候两者适合的场景不同。`sorted()`函数是在Python2.4版本中引入的，在这之前只有`sort()`函数。`sorted()`函数会返回一个排序后的列表，原有列表保持不变；而`sort()`函数会直接修改原有列表，函数返回为`None`。来看下面的例子：

```
>>> a=['1',1,'a',3,7,'n']
>>> sorted(a)
[1, 3, 7, '1', 'a', 'n']
>>> a
['1', 1, 'a', 3, 7, 'n']
>>> print a.sort()
None
>>> a
[1, 3, 7, '1', 'a', 'n']
>>>
```

因此如果实际应用过程中需要保留原有列表，使用`sorted()`函数较为适合，否则可以选择`sort()`函数，因为`sort()`函数不需要复制原有列表，消耗的内存较少，效率也较高。

3) 无论是`sort()`还是`sorted()`函数，传入参数`key`比传入参数`cmp`效率要高。`cmp`传入的函数在整个排序过程中会调用多次，函数开销较

大；而key针对每个元素仅作一次处理，因此使用key比使用cmp效率要高。下面的测试例子显示使用key比cmp约快50%。

```
>>> from timeit import Timer
>>> Timer(stmt="sorted(xs, key=lambda x: x[1])", setup="xs=range(100); xs=zip(xs, xs);").timeit(10000)
0.2900448249509081
>>>
>>> Timer(stmt="sorted(xs, cmp=lambda a, b: cmp(a[1], b[1]))", setup="xs=range(100); xs=zip(xs, xs);").timeit(10000)
0.47374972749250155
>>>
```

4) sorted() 函数功能非常强大，使用它可以方便地针对不同的数据结构进行排序，从而满足不同需求。来看下列例子。

·**对字典进行排序**：下面的例子中根据字典的值进行排序，即将phonebook对应的电话号码按照数字大小进行排序。

```
>>> phonebook = {'Linda': '7750', 'Bob': '9345', 'Carol': '5834'}
>>> from operator import itemgetter
>>> sorted_pb = sorted(phonebook.iteritems(), key=itemgetter(1))
>>> print sorted_pb
[('Carol', '5834'), ('Linda', '7750'), ('Bob', '9345')]
>>>
```

·**多维list排序**：实际情况下也会碰到需要对多个字段进行排序的情况，如根据学生的成绩、对应的等级依次排序。当然这在DB里面用SQL语句很容易做到，但使用多维列表联合sorted()函数也可以轻易达到类似的效果。

```
>>> from operator import itemgetter
>>> gameresult = [['Bob', 95.00, 'A'], ['Alan', 86.0, 'C'], ['Mandy', 82.5, 'A'], ['Rob', 86, 'E']] #
分别表示学生的姓名、成绩、等级
>>> sorted(gameresult, key=operator.itemgetter(2, 1))
[['Mandy', 82.5, 'A'], ['Bob', 95.0, 'A'], ['Alan', 86.0, 'C'], ['Rob', 86, 'E']] #
当第二个字段成绩相同的时候按照等级从低到高排序
]
```

·字典中混合list排序：如果字典中的key或者值为列表，需要对列表中的某一个位置的元素排序也是可以做到的。下面的例子中针对字典mydict的value结构[n,m]中的n按照从小到大的顺序排列。

```
>>> mydict = { 'Li': ['M',7],
...            'Zhang': ['E',2],
...            'Wang': ['P',3],
...            'Du': ['C',2],
...            'Ma': ['C',9],
...            'Zhe': ['H',7] }
>>>
>>> from operator import itemgetter
>>> sorted(mydict.iteritems(), key=lambda (k,v): operator.itemgetter(1)(v))
[('Zhang', ['E', 2]), ('Du', ['C', 2]), ('Wang', ['P', 3]), ('Li', ['M', 7]), ('Zhe', ['H', 7]), ('Ma', ['C', 9])]
```

·List中混合字典排序：如果列表中的每一个元素为字典形式，需要针对字典的多个key值进行排序也不难实现。下面的例子是针对list中的字典元素按照rating和name进行排序的实现方法。

```
>>> gameresult = [{ "name":"Bob", "wins":10, "losses":3, "rating":75.00 },
...                { "name":"David", "wins":3, "losses":5, "rating":57.00 },
...                { "name":"Carol", "wins":4, "losses":5, "rating":57.00 },
...                { "name":"Patty", "wins":9, "losses":3, "rating": 71.48 }]
>>> from operator import itemgetter
>>> sorted(gameresult , key=operator.itemgetter("rating","name"))
[{'wins': 4, 'losses': 5, 'name': 'Carol', 'rating': 57.0}, {'wins': 3, 'losses': 5, 'name': 'David', 'rating': 57.0}, {'wins': 9, 'losses': 3, 'name': 'Patty', 'rating': 71.48}, {'wins': 10, 'losses': 3, 'name': 'Bob', 'rating': 75.0}]
>>>
```

建议38：使用copy模块深拷贝对象

在正式讨论本节内容之前我们先来了解一下浅拷贝和深拷贝的概念：

·浅拷贝（**shallow copy**）：构造一个新的复合对象并将原对象中发现的引用插入该对象中。浅拷贝的实现方式有多种，如工厂函数、切片操作、**copy**模块中的**copy**操作等。

·深拷贝（**deep copy**）：也构造一个新的复合对象，但是遇到引用会继续递归拷贝其所指向的具体内容，也就是说它会针对引用所指向的对象继续执行拷贝，因此产生的对象不受其他引用对象操作的影响。深拷贝的实现需要依赖 **copy** 模块的**deepcopy()**操作。

下面我们通过一段简单的程序来说明浅拷贝和深拷贝之间的区别。

```
import copy
class Pizza(object):
    def __init__(self, name, size, price):
        self.name = name
        self.size = size
        self.price = price
    def getPizzaInfo(self):
        ①获取Pizza
        相关信息
        return self.name, self.size, self.price
    def showPizzaInfo(self):
        ②显示Pizza
        信息
        print "Pizza name:" + self.name
        print "Pizza size:" + str(self.size)
        print "Pizza price:" + str(self.price)
    def changeSize(self, size):
        self.size = size
    def changePrice(self, price):
        self.price = price
class Order(object):
    ③订单类
    def __init__(self, name):
        self.customername = name
        self.pizzaList = []
        self.pizzaList.append(Pizza("Mushroom", 12, 30))
```

```

def ordermore(self,pizza):
    self.pizzaList.append(pizza)
def changeName(self,name):
    self.customername=name
def getorderdetail(self):
    print "customer name:"+self.customername
    for i in self.pizzaList:
        i.showPizzaInfo()
def getPizza(self,number):
    return self.pizzaList[number]
customer1=Order("zhang")
customer1.ordermore(Pizza("seafood",9,40))
customer1.ordermore(Pizza("fruit",12,35))
print "customer1 order infomation:"
customer1.getorderdetail()
print "-----"

```

程序描述的是客户在**Pizza**店里下了一个订单，并将具体的订单信息打印出来的场景。运行输出结果如下：

```

customer name:zhang
Pizza name:Mushroom
Pizza size:12
Pizza price:30
Pizza name:seafood
Pizza size:9
Pizza price:40
Pizza name:fruit
Pizza size:12
Pizza price:35
-----

```

假设现在客户2也想下一个跟客户1一样的订单，只是要将预定的水果披萨的尺寸和价格进行相应的修改。于是服务员拷贝了客户1的订单信息并做了一定的修改，代码如下：

```

customer2=copy.copy(customer1)
print "order 2 customer name:"+customer2.customername
customer2.changeName("li")
customer2.getPizza(2).changeSize(9)
customer2.getPizza(2).changePrice(30)
print "customer2 order infomation:"
customer2.getorderdetail()
print "-----"

```

上面这段程序的输出也没有什么问题，完全满足了客户2的需求。输出结果如下所示：

```
order 2 customer name:zhang
customer2 order infomation:
customer name:li
Pizza name:Mushroom
Pizza size:12
Pizza price:30
Pizza name:seafood
Pizza size:9
Pizza price:40
Pizza name:fruit
Pizza size:9
Pizza price:30
-----
```

在修改完客户2的订单信息之后，现在我们来检查一下客户1的订单信息：

```
print "customer1 order information:"
customer1.getorderdetail()
```

你会发现客户1的订单内容除了客户姓名外，其他的居然和客户2的订单具体内容一样了。

```
-----
customer1 order infomation:
customer name:zhang
Pizza name:Mushroom
Pizza size:12
Pizza price:30
Pizza name:seafood
Pizza size:9
Pizza price:40
Pizza name:fruit
Pizza size:9
Pizza price:30
```

这是怎么回事呢？客户1根本没要求修改订单的内容，这样的结果必定会直接影响到客户满意度。问题出现在哪里？这是我们本节要重点讨论的内容。我们先来分析客户1和客户2订单内容的关系图，如图4-1所示。

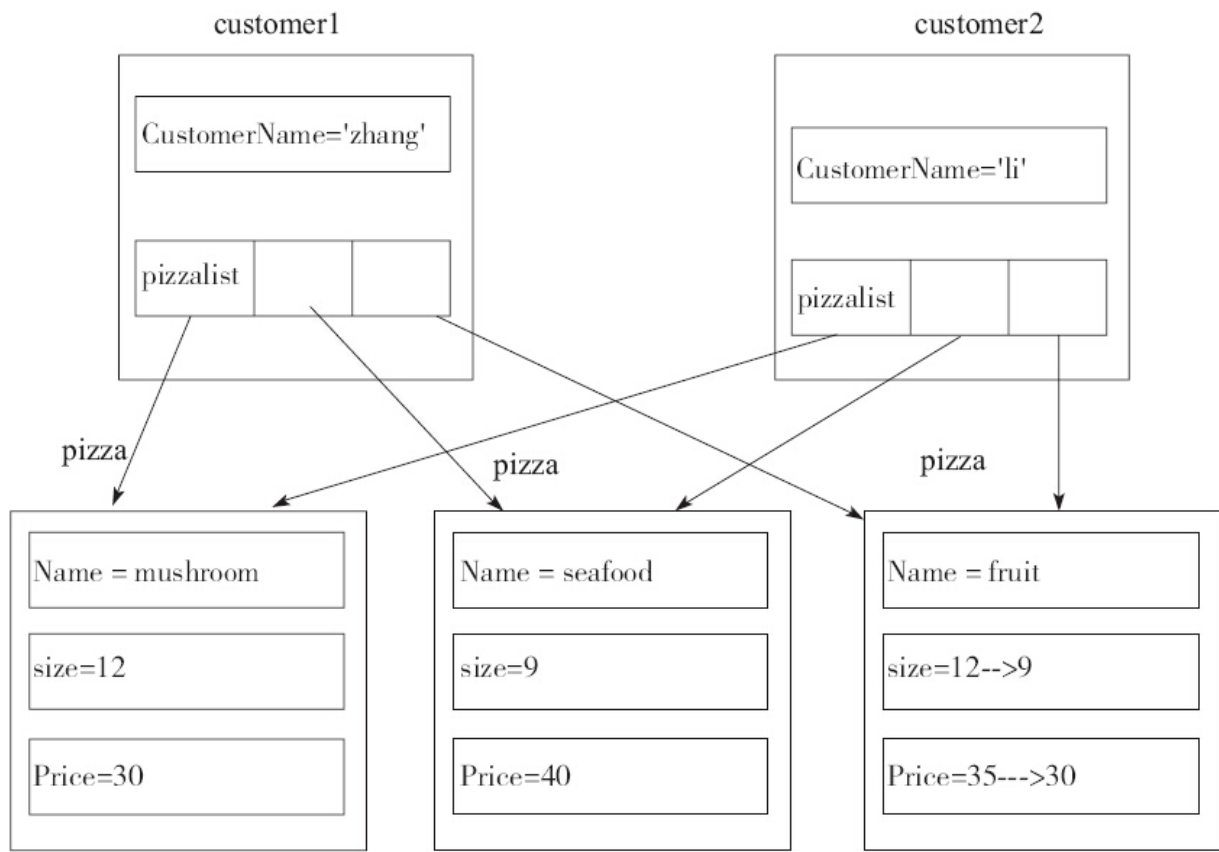


图4-1 客户1和客户2订单的关系示意图

customer1中的pizzalist是一个由Pizza对象组成的列表，其中存放的实际是对一个个具体Pizza对象的引用，在内存中就是一个具体的地址，可以通过查看id得到相关信息。

```

print id(customer1.pizzaList[0])
输出14099440
print id(customer1.pizzaList[1])
输出14101392
print id(customer1.pizzaList[2])
输出 14115344
print id(customer1.pizzaList)
输出 13914800

```

customer2的订单通过copy.copy(customer1) 获得，通过id函数查看customer2中pizzalist的具体Pizza对象你会发现它们和customer1中的输出是一样的（读者可以自行验证）。这是由于通过copy.copy()得到的

`customer2`是`customer1`的一个浅拷贝，它仅仅拷贝了`pizzalist`里面对象的地址而不对对应地址所指向的具体内容（即具体的`pizza`）进行拷贝，因此`customer2`中的`pizzaList`所指向的具体内容是和`customer1`中一样的，如图4-1所示。所以对`pizza fruit`的修改直接影响了`customer1`的订单内容。实际上在包含引用的数据结构中，浅拷贝并不能进行彻底的拷贝，当存在列表、字典等不可变对象的时候，它仅仅拷贝其引用地址。要解决上述问题需要用到深拷贝，深拷贝不仅拷贝引用也拷贝引用所指向的对象，因此深拷贝得到的对象和原对象是相互独立的。

上面的例子充分展示了浅拷贝和深拷贝之间的差异，在实际应用中要特别注意这两者之间的区别。实际上Python `copy`模块提供了与浅拷贝和深拷贝对应的两种方法的实现，通过名字便可以轻易进行区分，模块在拷贝出现异常的时候会抛出`copy.error`。

```
copy.copy(x)
: Return a shallow copy of x.
copy.deepcopy(x)
: Return a deep copy of x.
exception copy.error
: Raised for module specific errors.
```

实际上，上面的程序应该将`customer2=copy.copy(customer1)`改为`copy.deepcopy()`来实现（请读者自行验证）。关于对象拷贝读者可以查看网页http://en.wikipedia.org/wiki/Object_copy进行阅读扩展。

建议39：使用Counter进行计数统计

计数统计相信大家都不陌生，简单地说就是统计某一项出现的次数。实际应用中很多需求都需要用到这个模型，如检测样本中某一值出现的次数、日志分析某一消息出现的频率、分析文件中相同字符串出现的概率等。这种类似的需求有很多种实现方法。我们逐一起来看一下使用不同数据结构时的实现方式。

1) 使用dict。

```
>>> some_data = ['a','2',2,4,5,'2','b',4,7,'a',5,'d','a','z']
>>> count_frq = dict()
>>> for item in some_data:
...     if item in count_frq:
...         count_frq[item] +=1
...     else:
...         count_frq[item] = 1
...
>>> print count_frq
{'a': 3, 2: 1, 'b': 1, 4: 2, 5: 2, 7: 1, '2': 2, 'z': 1, 'd': 1}
```

2) 使用defaultdict。

```
>>> from collections import defaultdict
>>> some_data = ['a','2',2,4,5,'2','b',4,7,'a',5,'d','a','z']
>>> count_frq = defaultdict(int)
>>> for item in some_data:
...     count_frq[item] += 1
...
>>> print count_frq
defaultdict(<type 'int'>, {'a': 3, 2: 1, 'b': 1, 4: 2, 5: 2, 7: 1, '2': 2, 'z': 1, 'd': 1})
```

3) 使用set和list。

```
>>> some_data = ['a','2',2,4,5,'2','b',4,7,'a',5,'d','a','z']
>>> count_set = set(some_data)
>>> count_list = []
>>> for item in count_set:
...     count_list.append((item,some_data.count(item)))
```

```
...
>>> print count_list
[('a', 3), (2, 1), ('b', 1), (4, 2), (5, 2), (7, 1), ('2', 2), ('z', 1), ('d', 1)]
```

上面的方法都比较简单，但有没有更优雅、更Pythonic的解决方法呢？答案是使用collections.Counter。

```
>>> from collections import Counter
>>> some_data = ['a', '2', 2, 4, 5, '2', 'b', 4, 7, 'a', 5, 'd', 'a', 'z']
>>> print Counter(some_data)
Counter({'a': 3, 4: 2, 5: 2, '2': 2, 2: 1, 'b': 1, 7: 1, 'z': 1, 'd': 1})
```

Counter类是自Python2.7起增加的，属于字典类的子类，是一个容器对象，主要用来统计散列对象，支持集合操作+、-、&、|，其中&和|操作分别返回两个Counter对象各元素的最小值和最大值。它提供了3种不同的方式来初始化：

```
Counter("success")           #
可迭代对象
Counter(s=3, c=2, e=1, u=1)   #
关键字参数
Counter({"s":3, "c":2, "u":1, "e":1}) #
字典
```

可以使用elements()方法来获取Counter中的key值。

```
>>> list(Counter(some_data).elements())
['a', 'a', 'a', 2, 'b', 4, 4, 5, 5, 7, '2', '2', 'z', 'd']
```

利用most_common()方法可以找出前N个出现频率最高的元素以及它们对应的次数。

```
>>> Counter(some_data).most_common(2)
[('a', 3), (4, 2)]
```

当访问不存在的元素时，默认返回为0而不是抛出KeyError异常。

```
>>> (Counter(some_data))['y']  
0
```

`update()`方法用于被统计对象元素的更新，原有Counter计数器对象与新增元素的统计计数值相加而不是直接替换它们。

`subtract()`方法用于实现计数器对象中元素统计值相减，输入和输出的统计值允许为0或者负数。

```
>>> c = Counter("success")      #Counter({'s': 3, 'c': 2, 'e': 1, 'u': 1})  
>>> c.update("successfully")    #'s': 3, 'c': 2, 'l': 2, 'u': 2, 'e': 1, 'f': 1,  
'y': 1  
>>> c                          #s  
的值为变为6  
, 为上面s  
中对应值的和  
Counter({'s': 6, 'c': 4, 'u': 3, 'e': 2, 'l': 2, 'f': 1, 'y': 1})  
>>> c.subtract("successfully")  
>>> c  
Counter({'s': 3, 'c': 2, 'e': 1, 'u': 1, 'f': 0, 'l': 0, 'y': 0})
```

建议40：深入掌握ConfigParser

几乎所有的应用程序真正运行起来的时候，都会读取一个或几个配置文件。配置文件的意义在于用户不需要修改代码，就可以改变应用程序的行为，让它更好地为应用服务。比如pylint就带有一个参数`--rcfile`用以指定配置文件，实现对自定义代码风格的检测。常见的配置文件格式有XML和ini等，其中在MS Windows系统上，ini文件格式用得尤其多，甚至操作系统的API也都提供了相关的接口函数来支持它。类似ini的文件格式，在Linux等操作系统中也是极常用的，比如pylint的配置文件就是这个格式。但凡这种常用的东西，Python都有个标准库来支持它，也就是ConfigParser。

ConfigParser的基本用法通过手册可以掌握，但是仍然有几个知识点值得在这里跟大家说一下。首先就是`getboolean()`这个函数。`getboolean()`根据一定的规则将配置项的值转换为布尔值，如以下的配置：

```
[section1]
option1=0
```

当调用`getboolean('section1', 'option1')`时，将返回False。不过`getboolean()`的真值规则值得一说：除了0之外，no、false和off都会被转义为False，而对应的1、yes、true和on则都被转义为True，其他值都会导致抛出ValueError异常。这样的设计非常贴心，使得我们能够在不同的场合使用yes/no、true/false、on/off等更切合自然语言语法的词汇，提升配置文件的可维护性。

除了`getboolean()`之外，还需要注意的是配置项的查找规则。首先，在`ConfigParser`支持的配置文件格式里，有一个`[DEFAULT]`节，当读取的配置项在不在指定的节里时，`ConfigParser`将会到`[DEFAULT]`节中查找。举个例子，有如下配置文件：

```
$ cat example.conf
[DEFAULT]
in_default = 'an option value in default'
[section1]
```

简单地编写一段程序尝试通过`section1`获取`in_default`的值。

```
$ cat readini.py
import ConfigParser
conf = ConfigParser.ConfigParser()
conf.read('example.conf')
print conf.get('section1', 'in_default')
```

运行结果如下：

```
$ python readini.py
'an option value in default'
```

可见`ConfigParser`的行为跟上文描述是一致的。不过，除此之外，还有一些机制导致项目对配置项的查找更复杂，这就是`class ConfigParser`构造函数中的`defaults`形参以及其`get(section, option[, raw[, vars]])`中的全名参数`vars`。如果把这些机制全部用上，那么配置项值的查找规则如下：

- 1) 如果找不到节名，就抛出`NoSectionError`。
- 2) 如果给定的配置项出现在`get()`方法的`vars`参数中，则返回`vars`参数中的值。

- 3) 如果在指定的节中含有给定的配置项，则返回其值。
- 4) 如果在[DEFAULT]中有指定的配置项，则返回其值。
- 5) 如果在构造函数的defaults参数中有指定的配置项，则返回其值。
- 6) 抛出NoOptionError。

因为篇幅所限，这里就不提供示例了，大家可以自行构造相应的例子来验证。接下来要讲的是第三个特点。大家知道，在Python中字符串格式化可以使用以下语法：

```
>>> '%(protocol)s://%(server)s:%(port)s/' % { 'protocol':'http',
'server':'example.com',
      'port':1080}
'http://example.com:1080/'
```

其实ConfigParser支持类似的用法，所以在配置文件中可以使用。如，有如下配置选项：

```
$ cat format.conf
[DEFAULT]
conn_str = %(dbn)s://%(user)s:%(pw)s@%(host)s:%(port)s/%(db)s
dbn = mysql
user = root
host=localhost
port = 3306
[db1]
user = aaa
pw=ppp
db=example
[db2]
host=192.168.0.110
pw=www
db=example
```

这是一个很常见的SQLAlchemy应用程序的配置文件，通过这个配置文件能够获取不同的数据库配置相应的连接字符串，即conn_str。如你所见，conn_str定义在[DEFAULT]中，但当它通过不同的节名来获

取格式化后的值时，根据不同配置，得到不同的值。先来看以下代码：

```
$ cat readformatini.py
import ConfigParser
conf = ConfigParser.ConfigParser()
conf.read('format.conf')
print conf.get('db1', 'conn_str')
print conf.get('db2', 'conn_str')
```

非常简单的代码，就是通过`ConfigParser`获取`db1`和`db2`两个节的配置。然后看如下输出：

```
$ python readformatini.py
mysql://aaa:ppp@localhost:3306/example
mysql://root:www@192.168.0.110:3306/example
```

可以看到，当通过不同的节名调用`get()`方法时，格式化`conn_str`的参数是不同的，这个规则跟上述查找配置项的规则相同。

建议41：使用argparse处理命令行参数

尽管应用程序通常能够通过配置文件在不修改代码的情况下改变行为，但提供灵活易用的命令行参数依然非常有意义，比如：减轻用户的学习成本，通常命令行参数的用法只需要在应用程序名后面加`--help`参数就能获得，而配置文件的配置方法通常需要通过通读手册才能掌握；同一个运行环境中存在多个配置文件，那么需要通过命令行参数指定当前使用哪一个配置文件，如`pylint`的`--rcfile`参数就是做这个事的。

为了做好命令行处理这件事，Pythonista尝试好几个方案，标准库中留下的`getopt`、`optparse`和`argparse`就是证明。其中`getopt`是类似UNIX系统中`getopt()`这个C函数的实现，可以处理长短配置项和参数。如有命令行参数`-a -b -cfoo -d bar a1 a2`，在处理之后的结果是两个列表，其中一个为配置项列表`[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]`，每一个元素都由配置项名和其值（默认为空字符串）组成；另一个是参数列表`['a1', 'a2']`，每一个元素都是一个参数值。`getopt`的问题在于两点，一个是长短配置项需要分开处理，二是对非法参数和必填参数的处理需要手动。如：

```
try:
    opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
except getopt.GetoptError as err:
    print str(err) #
    此处输出类似 "option -a not recognized"
    的出错信息
    usage()
    sys.exit(2)
output = None
verbose = False
for o, a in opts:
    if o == "-v":
        verbose = True
    elif o in ("-h", "--help"):
        usage()
        sys.exit()
    elif o in ("-o", "--output"):
```

```
        output = a
    else:
        assert False, "unhandled option"
```

从for循环处可以看到，这种处理非常原始和不便，而从getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])函数调用时的"ho:v"和 ["help", "output="]两个实参可以看出，要编写和维护还是比较困难的，所以optparse就登场了。optparse比getopt要更加方便、强劲，与C风格的getopt不同，它采用的是声明式风格，此外，它能够自动生成应用程序的帮助信息。下面是一个例子：

```
from optparse import OptionParser
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")
(options, args) = parser.parse_args()
```

可以看到add_option()方法非常强大，同时支持长短配置项，还有默认值、帮助信息等，简单的几行代码，可以支持非常丰富的命令行接口。如，以下几个都是合法的应用程序调用：

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

除此之外，虽然没有声明帮助参数，但默认给加上了-h或--help支持，通过这两个参数调用应用程序，可以看到自动生成的帮助信息。

```
Usage: <yourscript> [options]
Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet            don't print status messages to stdout
```

不过 `optparse` 虽然很好，但是后来出现的 `argparse` 在继承了它声明式风格的优点之外，又多了更丰富的功能，所以现阶段最好用的参数处理标准库是 `argparse`，使 `optparse` 成为了一个被弃用的库。

因为 `argparse` 自 `optparse` 脱胎而来，所以用法倒也大致相同，都是先生成一个 `parser` 实例，然后增加参数声明。如上文中 `getopt` 的那个例子，可以用其改造为如下形式：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-o', '--output')
parser.add_argument('-v', dest='verbose', action='store_true')
args = parser.parse_args()
```

可以看到，代码大大地减化了，代码更少，`bug` 更少。与 `optparse` 中的 `add_option()` 类似，`add_argument()` 方法用以增加一个参数声明。与 `add_option()` 相比，它有几个方面的改进，其中之一就是支持类型增多，而且语法更加直观。表现在 `type` 参数的值不再是一个字符串，而是一个可调用对象，比如在 `add_option()` 调用时是 `type="int"`，而在 `add_argument()` 调用时直接写 `type=int` 就可以了。除了支持常规的 `int/float` 等基本数值类型外，`argparse` 还支持文件类型，只要参数合法，程序就能够使用相应的文件描述符。如：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<open file 'out.txt', mode 'w' at 0x...>)
```

另外，扩展类型也变得更加容易，任何可调用对象，比如函数，都可以作为 `type` 的实参。与 `type` 类似，`choices` 参数也支持更多的类型，而不是像 `add_option` 那样只有字符串。比如下面这句代码是合法的：

```
parser.add_argument('door', type=int, choices=range(1, 4))
```

此外，`add_argument()`提供了对必填参数的支持，只要把`required`参数设置为`True`传递进去，当缺失这一参数时，`argparse`就会自动退出程序，并提示用户。

如果仅仅是`add_argument()`比`add_option()`更加强大一点，并不足以让它把`optparse`踢出标准库，`ArgumentParser`还支持参数分组。

`add_argument_group()`可以在输出帮助信息时更加清晰，这在用法复杂的CLI应用程序中非常有帮助，比如`setuptools`配套的`setup.py`文件，如果运行`python setup.py help`可以看到它的参数是分组的。下面是一个简单的示例：

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo
group1:
  group1 description
  foo      foo help
group2:
  group2 description
  --bar BAR  bar help
```

如果仅仅是更加漂亮的帮助信息输出不够吸引你，那么`add_mutually_exclusive_group(required=False)`就非常实用：它确保组中的参数至少有一个或者只有一个（`required=True`）。

`argparse`也支持子命令，比如`pip`就有`install/uninstall/freeze/list/show`等子命令，这些子命令又接受不同的参数，使用`ArgumentParser.add_subparsers()`就可以实现类似的功能。

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('--bar', type=int, help='bar help')
>>> parser.parse_args(['a', '--bar', '1'])
Namespace(bar=1)
```

看，就是这么简单！除了参数处理之外，当出现非法参数时，用户还需要做一些处理，处理完成后，一般是输出提示信息并退出应用程序。ArgumentParser提供了两个方法函数，分别是exit(status=0, message=None)和error(message)，可以省了import sys再调用sys.exit()的步骤。



注意

虽然argparse已经非常好用，但是上进的Pythonista并没有止步，所以他们发明了docopt，可以认为，它是比argparse更先进更易用的命令行参数处理器。它甚至不需要编写代码，只要编写类似argparse输出的帮助信息即可。这是因为它根据常见的帮助信息定义了一套领域特定语言（DSL），通过这个DSL Parser参数生成处理命令行参数的代码，从而实现对命令行参数的解释。因为docopt现在还不是标准库，所以在此不多介绍，有兴趣的读者可以自行去其官网（<http://docopt.org/>）学习。

建议42：使用pandas处理大型CSV文件

CSV（Comma Separated Values）作为一种逗号分隔型值的纯文本格式文件，在实际应用中经常用到，如数据库数据的导入导出、数据分析中记录的存储等。因此很多语言都提供了对CSV文件处理的模块，Python也不例外，其模块csv提供了一系列与CSV处理相关的API。我们先来看一下其中几个常见的API：

1) `reader(csvfile[, dialect='excel'][, fmtparam])`，主要用于CSV文件的读取，返回一个reader对象用于在CSV文件内容上进行行迭代。

参数`csvfile`，需要是支持迭代（Iterator）的对象，通常对文件（file）对象或者列表（list）对象都是适用的，并且每次调用`next()`方法的返回值是字符串（string）；参数`dialect`的默认值为`excel`，与`excel`兼容；`fmtparam`是一系列参数列表，主要用于需要覆盖默认的Dialect设置的情形。当`dialect`设置为`excel`的时候，默认Dialect的值如下：

```
class excel(Dialect):
    delimiter = ','          #
    单个字符，用于分隔字段
    quotechar = '"'          #
    用于对特殊符号加引号，常见的引号为"
    doublequote = True      #
    用于控制quotechar
    符号出现的时候的表现形式
    skipinitialspace = False #
    设置为true
    的时候delimiter
    后面的空格将会忽略
    lineterminator = '\r\n' #
    行结束符
    quoting = QUOTE_MINIMAL #
    是否在字段前加引号，QUOTE_MINIMAL
    表示仅当一个字段包
    含引号或者定义符号的时候才加引号
```

2) `csv.writer(csvfile, dialect='excel', **fmtparams)`，用于写入CSV文件。参数同上。来看一个使用例子。

```
with open('data.csv', 'wb') as csvfile:
    csvwriter = csv.writer(csvfile, dialect='excel', delimiter="|", quotechar='"',
        quoting=csv.QUOTE_MINIMAL)
    csvwriter.writerow(["1/3/09 14:44", "'Product1'", "1200'", "Visa", "Gouya"])
#
```

写入行

输出形式为: 1/3/09 14:44|'Product1'|1200'|Visa|Gouya

3) `csv.DictReader(csvfile, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwargs)`, 同`reader()`方法类似, 不同的是将读入的信息映射到一个字典中去, 其中字典的`key`由`fieldnames`指定, 该值省略的话将使用CSV文件第一行的数据作为`key`值。如果读入行的字段的个数大于`fieldnames`中指定的个数, 多余的字段名将会存放在`restkey`中, 而`restval`主要用于当读取行的域的个数小于`fieldnames`的时候, 它的值将会被用作剩下的`key`对应的值。

4) `csv.DictWriter(csvfile, fieldnames, restval="", extrasaction='raise', dialect='excel', *args, **kwargs)`, 用于支持字典的写入。

```
import csv
#DictWriter
with open('test.csv', 'wb') as csv_file:
#
    设置列名称
    FIELDS = ['Transaction_date', 'Product', 'Price', 'Payment_Type']
    writer = csv.DictWriter(csv_file, fieldnames=FIELDS)
    #
    写入列名称
    writer.writerow(dict(zip(FIELDS, FIELDS)))
    d = {'Transaction_date': '1/2/09 6:17', 'Product': 'Product1', 'Price': '1200',
        'Payment_Type': 'Mastercard'}
    #
    写入一行    Writer.writerow(d)
with open('test.csv', 'rb') as csv_file:
    for d in csv.DictReader(csv_file):
        print d
    #output d is: {'Product': 'Product1', 'Transaction_date': '1/2/09 6:17',
        'Price': '1200', 'Payment_Type': 'Mastercard'}
```

`csv`模块使用非常简单, 基本可以满足大部分需求。但你有没有思考过这个问题: 有些应用中需要解析和处理的CSV文件可能有上百MB甚至几个GB, 这种情况下`csv`模块是否能够应付呢? 先来做个实验, 临

时创建一个1GB的CSV文件并将其加载到内存中，看看会有什么问题发生。

```
>>> f = open('large.csv', 'wb')
>>> f.seek(1073741824-1)                                     #
创建大文件的技巧
>>> f.write("\0")
>>> f.close()
>>> import os
>>> os.stat("large.csv").st_size                               #
输出文件的大小
1073741824L
>>> with open("large.csv", "rb") as csvfile:
...     mycsv = csv.reader(csvfile, delimiter=';')
...     for row in mycsv:
...         print row
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
MemoryError                                                  #
发生了内存异常
>>>
```

上面的例子中当企图读入这个CSV文件的时候抛出了MemoryError异常。这是为什么？因为csv模块对于大型CSV文件的处理无能为力。这种情况下就需要考虑其他解决方案了，pandas模块便是较好的选择。

Pandas即Python Data Analysis Library，是为了解决数据分析而创建的第三方工具，它不仅提供了丰富的数据模型，而且支持多种文件格式处理，包括CSV、HDF5、HTML等，能够提供高效的大型数据处理。其支持的两种数据结构——Series和DataFrame——是数据处理的基础。下面先来介绍这两种数据结构。

·Series：它是一种类似数组的带索引的一维数据结构，支持的类型与NumPy兼容。如果不指定索引，默认为0到N-1。通过obj.values()和obj.index()可以分别获取值和索引。当给Series传递一个字典的时候，Series的索引将根据字典中的键排序。如果传入字典的时候同时重新指定了index参数，当index与字典中的键不匹配的时候，会出现数据丢失的情况，标记为NaN。

在pandas中用函数isnull()和notnull()来检测数据是否丢失。

```
>>> obj1 = Series([1, 'a', (1,2), 3], index=['a', 'b', 'c', 'd'])
>>> obj1#value
和index
一一匹配
a      1
b      a
c    (1, 2)
d      3
dtype: object
>>> obj2=Series({"Book":"Python","Author":"Dan","ISBN":"011334","Price":25},index=['book','Author','ISBN','Price'])
>>> obj2.isnull()
book      True      #
指定的index
与字典的键不匹配, 发生数据丢失
Author    False
ISBN      True      #
指定的index
与字典的键不匹配, 发生数据丢失
Price     False
dtype: bool
>>>
```

·**DataFrame**: 类似于电子表格, 其数据为排好序的数据列的集合, 每一列都可以是不同的数据类型, 它类似于一个二维数据结构, 支持行和列的索引。和**Series**一样, 索引会自动分配并且能根据指定的列进行排序。使用最多的方式是通过一个长度相等的列表的字典来构建。构建一个**DataFrame**最常用的方式是用一个相等长度列表的字典或NumPy数组。**DataFrame**也可以通过**columns**指定序列的顺序进行排序。

```
>>> data = {'OrderDate': ['1-6-10', '1-23-10', '2-9-10', '2-26-10', '3-15-10'],
...         'Region': ['East', 'Central', 'Central', 'West', 'East'],
...         'Rep': ['Jones', 'Kivell', 'Jardine', 'Gill', 'Sorvino']}
>>>
>>> DataFrame(data,columns=['OrderDate','Region','Rep'])#
通过字典构建, 按照columns
指定的顺序排序
   OrderDate  Region  Rep
0    1-6-10    East  Jones
1    1-23-10  Central  Kivell
2    2-9-10   Central  Jardine
3    2-26-10    West   Gill
4    3-15-10    East  Sorvino
```

Pandas中处理CSV文件的函数主要为read_csv()和to_csv()这两个，其中read_csv()读取CSV文件的内容并返回DataFrame，to_csv()则是其逆过程。两个函数都支持多个参数，由于其参数众多且过于复杂，本节不对各个参数一一介绍，仅选取几个常见的情形结合具体例子介绍。下面举例说明，其中需要处理的CSV文件格式如图4-2所示。

```
OrderDate,Region,Rep,Item,Units,Unit Cost,Total
1-6-10,East,Jones,Pencil,95, 1.99 , 189.05
1-23-10,Central,Kivell,Binder,50, 19.99 , 999.50
2-9-10,Central,Jardine,Pencil,36, 4.99 , 179.64
2-26-10,Central,Gill,Pen,27, 19.99 , 539.73
3-15-10,West,Sorvino,Pencil,56, 2.99 , 167.44
4-1-10,East,Jones,Binder,60, 4.99 , 299.40
4-18-10,Central,Andrews,Pencil,75, 1.99 , 149.25
```

图4-2 CSV文件示例

1) 指定读取部分列和文件的行数。具体的实现代码如下：

```
>>> df = pd.read_csv("SampleData.csv",nrows=5,usecols=['OrderDate','Item','Total'])
>>> df
  OrderDate  Item  Total
0  1-6-10  Pencil  189.05
1  1-23-10  Binder  999.50
2  2-9-10  Pencil  179.64
3  2-26-10    Pen  539.73
4  3-15-10  Pencil  167.44
```

方法read_csv()的参数nrows指定读取文件的行数，usecols指定所要读取的列的列名，如果没有列名，可直接使用索引0、1、...、n-1。上述两个参数对大文件处理非常有用，可以避免读入整个文件而只选取所需要部分进行读取。

2) 设置CSV文件与excel兼容。dialect参数可以是string也可以是csv.Dialect的实例。如果将图4-2所示的文件格式改为使用“|”分隔符，则

需要设置dialect相关的参数。error_bad_lines设置为False，当记录不符合要求的时候，如记录所包含的列数与文件列设置不相等时可以直接忽略这些列。下面的代码用于设置CSV文件与excel兼容，其中分隔符为“|”，而error_bad_lines=False会直接忽略不符合要求的记录。

```
>>> dia = csv.excel()
>>> dia.delimiter="|" #
设置分隔符
>>> pd.read_csv("SD.csv")
  OrderDate|Region|Rep|Item|Units|Unit Cost|Total
0      1-6-10|East|Jones|Pencil|95|1.99 |189.05
1  1-23-10|Central|Kivell|Binder|50|19.99 |999.50...
>>> pd.read_csv("SD.csv",dialect = dia,error_bad_lines=False)
Skipping line 3: expected 7 fields, saw 10 #
所有不符合格式要求的列将直接忽略
  OrderDate Region  Rep  Item  Units  Unit Cost  Total
0      1-6-10   East  Jones  Pencil    95    1.99  189.05
>>>
```

3) 对文件进行分块处理并返回一个可迭代的对象。分块处理可以避免将所有的文件载入内存，仅在使用的时候读入所需内容。参数chunksize设置分块的文件行数，10表示每一块包含10个记录。将参数iterator设置为True时，返回值为TextFileReader，它是一个可迭代对象。来看下面的例子，当chunksize=10、iterator=True时，每次输出为包含10个记录的块。

```
>>> reader = pd.read_table("SampleData.csv",chunksize=10,iterator=True)
>>> reader
<pandas.io.parsers.TextFileReader object at 0x0314BE70>
>>> iter(reader).next() #
将TextFileReader
转换为迭代器并调用next
方法
  OrderDate,Region,Rep,Item,Units,Unit Cost,Total #
每次读入10
行
0      1-6-10,East,Jones,Pencil,95, 1.99 , 189.05
1  1-23-10,Central,Kivell,Binder,50, 19.99 , 999.50
2    2-9-10,Central,Jardine,Pencil,36, 4.99 , 179.64
3      2-26-10,Central,Gill,Pen,27, 19.99 , 539.73
4      3-15-10,West,Sorvino,Pencil,56, 2.99 , 167.44
5      4-1-10,East,Jones,Binder,60, 4.99 , 299.40
6  4-18-10,Central,Andrews,Pencil,75, 1.99 , 149.25
7    5-5-10,Central,Jardine,Pencil,90, 4.99 , 449.10
8    5-22-10,West,Thompson,Pencil,32, 1.99 , 63.68
9      6-8-10,East,Jones,Binder,60, 8.99 , 539.40
```

4) 当文件格式相似的时候，支持多个文件合并处理。以下例子用于将3个格式相同的文件进行合并处理。

```
>>> filelst = os.listdir("test")
>>> print filelst                                     #
同时存在3
个格式相同的文件
['s1.csv', 's2.csv', 's3.csv']
>>> os.chdir("test")
>>> dfs =[pd.read_csv(f) for f in filelst]
>>> total_df = pd.concat(dfs)                         #
将文件合并
>>> total_df
```

	OrderDate	Region	Rep	Item	Units	Unit Cost	Total
0	1-6-10	East	Jones	Pencil	95	1.99	189.05
1	1-23-10	Central	Kivell	Binder	50	19.99	999.5

了解完pandas后，读者可以自行实验一下使用pandas处理前面生成的1GB的文件，看看还会不会抛出MemoryError异常。

在处理CSV文件上，特别是大型CSV文件，pandas不仅能够做到与csv模块兼容，更重要的是其CSV文件以DataFrame的格式返回，pandas对这种数据结构提供了非常丰富的处理方法，同时pandas支持文件的分块和合并处理，非常灵活，由于其底层很多算法采用Cython实现运行速度较快。实际上pandas在专业的数据处理与分析领域，如金融等行业已经得到广泛的应用。

建议43： 一般情况使用ElementTree解析XML

`xml.dom.minidom`和`xml.sax`大概是Python中解析XML文件最广为人知的两个模块了，原因一是这两个模块自Python 2.0以来就成为Python的标准库；二是网上关于这两个模块的使用方面的资料最多。作为主要解析XML方法的两种实现，DOM需要将整个XML文件加载到内存中并解析为一棵树，虽然使用较为简单，但占用内存较多，性能方面不占优势，并且不够Pythonic；而SAX是基于事件驱动的，虽不需要全部装入XML文件，但其处理过程却较为复杂。实际上Python中对XML的处理还有更好的选择，ElementTree便是其中一个，一般情况下使用ElementTree便已足够。它从Python2.5开始成为标准模块，cElementTree是ElementTree的Cython实现，速度更快，消耗内存更少，性能上更占优势，在实际使用过程中应该尽量优先使用cElementTree。由于两者使用方式上完全兼容本文将两者看做一个物件，除非说明不再刻意区分。ElementTree在解析XML文件上具有以下特性：

- 使用简单。它将整个XML文件以树的形式展示，每一个元素的属性以字典的形式表示，非常方便处理。

- 内存上消耗明显低于DOM解析。由于ElementTree底层进行了一定的优化，并且它的iterparse解析工具支持SAX事件驱动，能够以迭代的形式返回XML部分数据结构，从而避免将整个XML文件加载到内存中，因此性能上更优化，相比于SAX使用起来更为简单明了。

- 支持XPath查询，非常方便获取任意节点的值。

这里需要说明的是，一般情况指的是：XML文件大小适中，对性能要求并非非常严格。如果在实际过程中需要处理的XML文件大小在GB或近似GB级别，第三方模块lxml会获得较优的处理结果。关于lxml模块的介绍请参考本章后续小节或者参考文章“使用由Python编写的lxml实现高性能XML解析”，可通过链接<http://www.ibm.com/developerworks/cn/xml/x-hiperfparse/>可以访问。

下面结合具体的实例来说明elementtree解析XML文件常用的方法。需要解析的XML实例如下：

```
<systems>
  <system platform="linux" name="linuxtest">
    <purpose>automation test</purpose>
    <system_type>virtual</system_type>
    <ip_address/>
    <commands_on_boot>
      <command_details>
        <!-- Set root password. -->
        <command>echo root:mytestpwd | sudo /usr/
          sbin/chpasswd</command>
        <userid>root2</userid>
        <password>Passw0rd</password>
      </command_details>
      <command_details>
        <command>mkdir /TEST; chmod 777
/TEST</command>
      </command_details>
    </commands_on_boot>
  </system>
  <system platform="aix" name="aixtest">
    <purpose>manual test</purpose>
    <system_type>virtual</system_type>
    <ip_address/>
    <commands_on_boot>
      <command_details>
        <!-- Set root password. -->
        <command>echo root:mytestpwd | sudo /usr/
          sbin/chpasswd</command>
        <userid>root2</userid>
        <password>Passw0rd</password>
      </command_details>
      <command_details>
        <command>mkdir /TEST; chmod 777
/TEST</command>
      </command_details>
    </commands_on_boot>
  </system>
</systems>
```

模块ElementTree主要存在两种类型ElementTree和Element，它们支持的方法以及对应的使用示例如表4-1和表4-2所示。

表4-1 ElementTree主要的方法和使用示例

主要的方法、属性	方法说明以及示例
getroot()	返回 xml 文档的根节点 >>> import xml.etree.ElementTree as ET >>> tree = ET.ElementTree(file="test.xml") >>> root = tree.getroot() >>> print root <Element 'systems' at 0x26cbff0> >>> print root.tag systems
find(match) findall(match) findtext(match, default=None)	同 Element 相关的方法类似，只是从跟节点开始搜索（见表 4-2） >>> for i in root.findall("system/purpose"): ... print i.text ... automation test manual test >>> print root.findtext("system/purpose") automation test >>> print root.find("system/purpose") <Element 'purpose' at 0x26d2170>
iter(tag=None)	从 xml 根结点开始，根据传入的元素的 tag 返回所有的元素集合的迭代器 >>> for i in tree.iter(tag="command"): ... print i.text ... echo root:mytestpwd sudo /usr/sbin/chpasswd mkdir /TEST; chmod 777 /TEST echo root:mytestpwd sudo /usr/sbin/chpasswd mkdir /TEST; chmod 777 /TEST

(续)

主要的方法、属性	方法说明以及示例
iterfind(match)	根据传入的 tag 名称或者 path 以迭代器的形式返回所有的子元素 >>> for i in tree.iterfind("system/purpose"): ... print i.text ... automation test manual test

表4-2 Element主要的方法和使用示例

主要的方法、属性	方法说明以及示例
tag	字符串，用来表示元素所代表的名称 >>> print root[1].tag # 输出 system
text	表示元素所对应的具体值 >>> print root[1].text # 输出空串
attrib	用字典表示的元素的属性 >>> print root[1].attrib # 输出 {'platform': 'aix', 'name': 'aixtest'}
get(key, default=None)	根据元素属性字典的 key 值获取对应的值，如果找不到对应的属性，则返回 default >>> print root[1].attrib.get("platform") # 输出 aix
items()	将元素属性以 (名称, 值) 的形式返回 >>> print root[1].items() # [('platform', 'aix'), ('name', 'aixtest')]
keys()	返回元素属性的 key 值集合 >>> print root[1].keys() # 输出 ['platform', 'name']
find(match)	根据传入的 tag 名称或者 path 返回第一个对应的 element 对象，或者返回 None
findall(match)	根据传入的 tag 名称或者 path 以列表的形式返回所有符合条件的元素
findtext(match, default=None)	根据传入的 tag 名称或者 path 返回第一个对应的 element 对象对应的值，即 text 属性，如果找不到则返回 default 的设置
list(elem)	根据传入的元素的名称返回其所有的子节点 >>> for i in list(root.findall("system/system_type")): ... print i.text ... # 输出 virtual virtural

前面我们提到elementree的iterparse工具能够避免将整个XML文件加载到内存，从而解决当读入文件过大内存而消耗过多的问题。iterparse返回一个可以迭代的由元组(时间,元素)组成的流对象，支持两个参数——source和events，其中event有4种选择——start、end、startns和endns（默认为end），分别与SAX解析的startElement、endElement、startElementNS和endElementNS一一对应。

本节最后来看一下iterparse的使用示例：统计userid在整个XML出现的次数。

```
>>> count = 0
>>> for event,elem in ET.iterparse("test.xml"):#
对iterparse
的返回值进行迭代
...     if event == 'end':
```

```
...         if elem.tag == 'userid':
...             count+=1
...         elem.clear()
...
>>> print count
2
```

建议44：理解模块pickle优劣

在实际应用中，序列化的场景很常见，如：在磁盘上保存当前程序的状态数据以便重启的时候能够重新加载；多用户或者分布式系统中数据结构的网络传输时，可以将数据序列化后发送给一个可信网络对端，接收者进行反序列化后便可以重新恢复相同的对象；`session`和`cache`的存储等。序列化，简单地说就是把内存中的数据结构在不丢失其身份和类型信息的情况下转成对象的文本或二进制表示的过程。对象序列化后的形式经过反序列化过程应该能恢复为原有对象。Python中有很多支持序列化的模块，如`pickle`、`json`、`marshal`和`shelve`等。最广为人知的为`pickle`，我们来仔细分析一下这个模块。

`pickle`估计是最通用的序列化模块了，它还有个C语言的实现`cPickle`，相比`pickle`来说具有较好的性能，其速度大概是`pickle`的1000倍，因此在大多数应用程序中应该优先使用`cPickle`（注：`cPickle`除了不能被继承之外，它们两者的使用基本上区别不大，除有特殊情况，本节将不再做具体区分）。`pickle`中最主要的两个函数对为`dump()`和`load()`，分别用来进行对象的序列化和反序列化。

·`pickle.dump(obj, file[, protocol])`：序列化数据到一个文件描述符（一个打开的文件、套接字等）。参数`obj`表示需要序列化的对象，包括布尔、数字、字符串、字节数组、`None`、列表、元组、字典和集合等基本数据类型，此外`pickle`还能够处理循环，递归引用对象、类、函数以及类的实例等。参数`file`支持`write()`方法的文件句柄，可以为真实的文件，也可以是`StringIO`对象等。`protocol`为序列化使用的协议版本，0表示ASCII协议，所序列化的对象使用可打印的ASCII码表示；1

表示老式的二进制协议；2表示2.3版本引入的新二进制协议，比以前的更高效。其中协议0和1兼容老版本的Python。protocol默认值为0。

·load(file): 表示把文件中的对象恢复为原来的对象，这个过程也被称为反序列化。

来看一下load()和dump()的示例。

```
>>> import cPickle as pickle
>>> my_data = {"name" : "Python", "type" : "Language", "version" : "2.7.5"}
>>> fp = open("picklefile.dat", "wb") #
打开要写入的文件
>>> pickle.dump(my_data, fp)          #
使用dump
进行序列化
>>> fp.close()
>>>
>>> fp = open("picklefile.dat", "rb")
>>> out = pickle.load(fp)             #
反序列化
>>> print(out)
{'version': '2.7.5', 'type': 'Language', 'name': 'Python'}
>>> fp.close()
```

pickle之所以能成为通用的序列化模块，与其良好的特性是分不开的，总结为以下几点：

1) 接口简单，容易使用。通过dump()和load()便可轻易实现序列化和反序列化。

2) pickle的存储格式具有通用性，能够被不同平台的Python解析器共享，比如，Linux下序列化的格式文件可以在Windows平台的Python解析器上进行反序列化，兼容性较好。

3) 支持的数据类型广泛。如数字、布尔值、字符串，只包含可序列化对象的元组、字典、列表等，非嵌套的函数、类以及通过类的__dict__或者__getstate__()可以返回序列化对象的实例等。

4) pickle模块是可以扩展的。对于实例对象，pickle在还原对象的时候一般是不调用__init__()函数的，如果要调用__init__()进行初始化，对于古典类可以在类定义中提供__getinitargs__()函数，并返回一个元组，当进行unpickle的时候，Python就会自动调用__init__()，并把__getinitargs__()中返回的元组作为参数传递给__init__()，而对于新式类，可以提供__getnewargs__()来提供对象生成时候的参数，在unpickle的时候以Class.__new__(Class, *arg)的方式创建对象。对于不可序列化的对象，如sockets、文件句柄、数据库连接等，也可以通过实现pickle协议来解决这些局限，主要是通过特殊方法__getstate__()和__setstate__()来返回实例在被pickle时的状态。来看以下示例：

```
import cPickle as pickle
class TextReader:
    def __init__(self, filename):
        self.filename = filename          #
文件名称
        self.file = open(filename)       #
打开文件的句柄
        self.postion = self.file.tell()  #
文件的位置
    def readline(self):
        line = self.file.readline()
        self.postion = self.file.tell()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.postion, line)
    def __getstate__(self):               #
记录文件被pickle
时候的状态
        state = self.__dict__.copy()     #
获取被pickle
时的字典信息
        del state['file']
        return state
    def __setstate__(self, state):        #
设置反序列化后的状态
        self.__dict__.update(state)
        file = open(self.filename)
        self.file = file
reader = TextReader("zen.txt")
print(reader.readline())
print(reader.readline())
s = pickle.dumps(reader)                #
在dumps
的时候会默认调用__getstate__
new_reader = pickle.loads(s)            #
在loads
的时候会默认调用__setstate__
print(new_reader.readline())
```

5) 能够自动维护对象间的引用，如果一个对象上存在多个引用，`pickle`后不会改变对象间的引用，并且能够自动处理循环和递归引用。

```
>>> a = ['a', 'b']
>>> b = a                                     #b
引用对象a
>>> b.append('c')
>>> p = pickle.dumps((a,b))
>>> a1,b1 = pickle.loads(p)
>>> a1
['a', 'b', 'c']
>>> b1
['a', 'b', 'c']
>>> a1.append('d')                             #
反序列化对a1
对象的修改仍然会影响到b1
>>> b1
['a', 'b', 'c', 'd']
```

但`pickle`使用也存在以下一些限制：

- `pickle`不能保证操作的原子性。 `pickle`并不是原子操作，也就是说在一个`pickle`调用中如果发生异常，可能部分数据已经被保存，另外如果对象处于深递归状态，那么可能超出Python的最大递归深度。递归深度可以通过`sys.setrecursionlimit()`进行扩展。

- `pickle`存在安全性问题。 Python的文档清晰地表明它不提供安全性保证，因此对于一个从不可信的数据源接收的数据不要轻易进行反序列化。由于`loads()`可以接收字符串作为参数，这意味着精心设计的字符串给入侵提供了一种可能。在Python解释器中输入代码`pickle.loads("cos\nsystem\n(S'dir'\ntR.")`便可查看当前目录下所有文件。如果将`dir`替换为其他更具有破坏性的命令将会带来安全隐患。如果要进一步提高安全性，用户可以通过继承类`pickle.Unpickler`并重写`find_class()`方法来实现。

- `pickle`协议是Python特定的，不同语言之间的兼容性难以保障。用Python创建的`pickle`文件可能其他语言不能使用，如Perl、PHP、Java

等。

建议45：序列化的另一个不错的选择——JSON

JSON（JavaScript Object Notation）是一种轻量级数据交换格式，它基于JavaScript编程语言的一个子集，于1999年12月成为一个完全独立于语言的文本格式。由于其格式使用了其他许多流行编程的约定，如C、C++、C#、Java、JS、Python等，加之其简单灵活、可读性和可操作性较强、易于解析和使用等特点，逐渐变得流行起来，甚至有代替XML的趋势。关于JSON和XML之间的优劣，一直有很多争论，本书并不打算对这两者之间的是是非非做详尽的分析（笔者的观点是两者各有所长，在相当长的时间里还会共存共荣），这里关注的是JSON用于序列化方面的优势。在进行详细讨论之前，我们先来看看Python语言中对JSON的支持现状。

Python中有一系列的模块提供对JSON格式的支持，如simplejson、cjson、yajl、ujson，自Python2.6后又引入了标准库JSON。简单来说cjson和ujson是用C来实现的，速度较快。据cjson的文档表述：其速率比纯Python实现的json模块大概要快250倍。yajl是Cpython版本的JSON实现，而simplejson和标准库JSON本质来说无多大区别，实际上Python2.6中的json模块就是simplejson减去对Python2.4、2.5的支持以充分利用最新的兼容未来的功能。不过相对于simplejson，标准库更新相对较慢，Python2.7.5中simplejson对应的版本为2.0.9，而最新的simplejson的版本为3.3.0。在实际应用过程中将这两者结合较好的做法是采用如下import方法。

```
try: import simplejson as json
except ImportError: import json
```

本节仍采用标准库JSON来做一些探讨。Python的标准库JSON提供的最常用的方法与pickle类似，dump/dumps用来序列化，load/loads用来反序列化。需要注意的json默认不支持非ASCII-based的编码，如load方法可能在处理中文字符时不能正常显示，则需要通过encoding参数指定对应的字符编码。在序列化方面，相比pickle，JSON具有以下优势：

1) 使用简单，支持多种数据类型。JSON文档的构成非常简单，仅存在以下两大数据结构。

- 名称/值对的集合。在各种语言中，它被实现为一个对象、记录、结构、字典、散列表、键列表或关联数组。

- 值的有序列表。在大多数语言中，它被实现为数组、向量、列表或序列。在Python中对应支持的数据类型包括字典、列表、字符串、整数、浮点数、True、False、None等。JSON中数据结构和Python中的转换并不是完全一一对应，存在一定的差异，读者可以自行查阅文档。Python中一个JSON文档可以分解为如图4-3所示形式。

2) 存储格式可读性更为友好，容易修改。相比于pickle来说，json的格式更加接近程序员的思维，修改和阅读上要容易得多。dumps()函数提供了一个参数indent使生成的json文件可读性更好，0意味着“每个值单独一行”；大于0的数字意味着“每个值单独一行并且使用这个数字的空格来缩进嵌套的数据结构”。但需要注意的是，这个参数是以文件大小变大为代价的。如图4-4展示的是这两种格式之间的对比，其中json.dumps()使用了indent参数输出。

3) json支持跨平台跨语言操作，能够轻易被其他语言解析，如Python中生成的json文件可以轻易使用JavaScript解析，互操作性更强，而pickle格式的文件只能在Python语言中支持。此外json原生的JavaScript支持，客户端浏览器不需要为此使用额外的解释器，特别适

用于Web应用提供快速、紧凑、方便的序列化操作。此外，相比于 pickle， json的存储格式更为紧凑，所占空间更小。

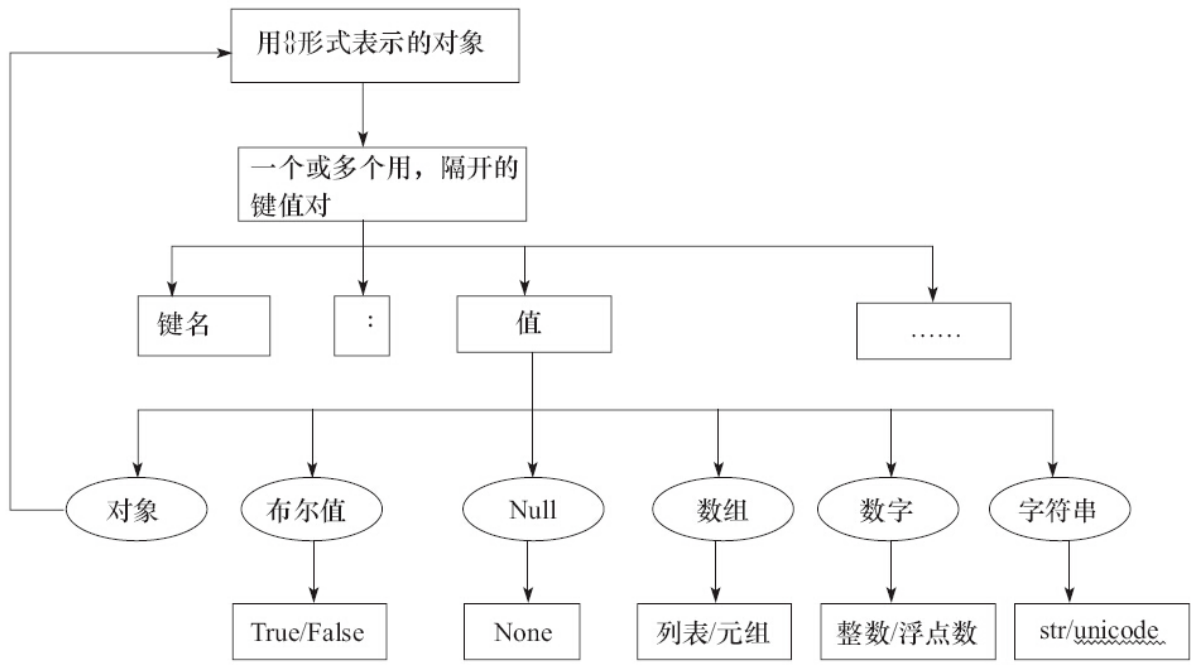


图4-3 json文档分解图

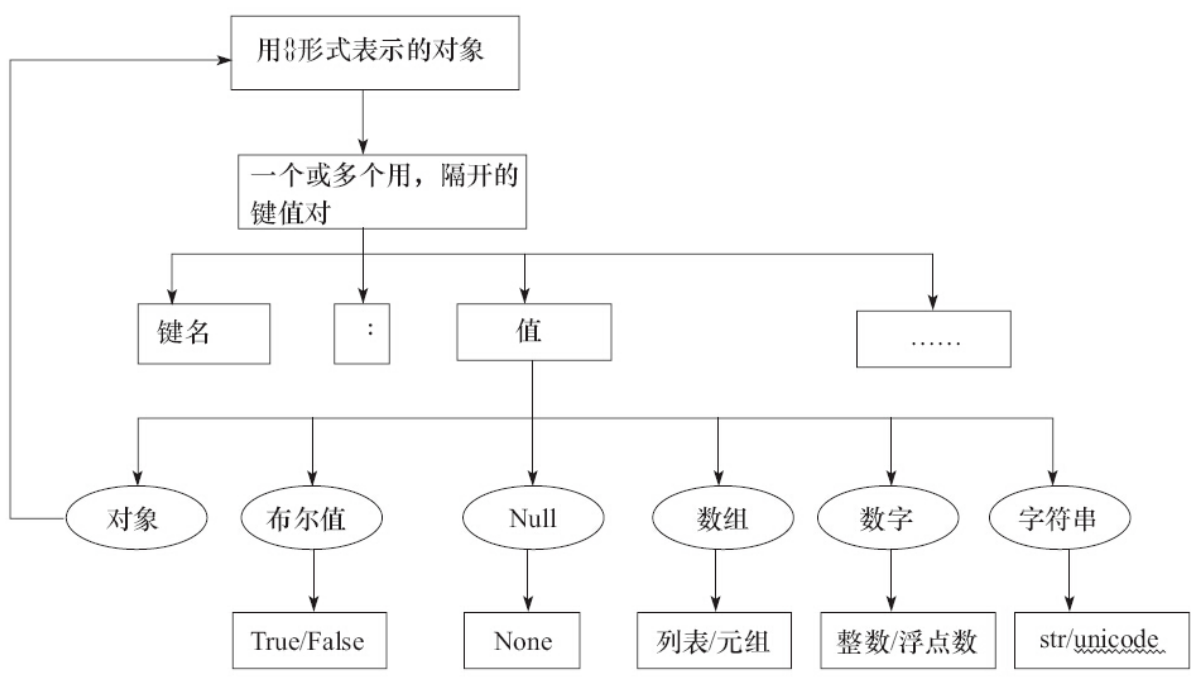


图4-4 pickle和json文件格式对比

4) 具有较强的扩展性。json模块还提供了编码（JSONEncoder）和解码类（JSONDecoder）以便用户对其默认不支持的序列化类型进行扩展。来看一个例子：

```
>>> d=datetime.datetime.now()
>>> d
datetime.datetime(2013, 9, 15, 8, 54, 59, 851000)
>>> json.dumps(d)
... ..
raise TypeError(repr(o) + " is not JSON serializable")
TypeError: datetime.datetime(2013, 9, 15, 8, 54, 59, 851000) is not JSON serializable
```

5) json在序列化datetime的时候会抛出TypeError异常，这是因为json模块本身不支持datetime的序列化，因此需要对json本身的JSONEncoder进行扩展。有多种方法可以实现，下面的例子是其中实现之一。

```
import datetime
from time import mktime
try: import simplejson as json
except ImportError: import json
class DateTimeEncoder(json.JSONEncoder):          #
对JSONEncoder
进行扩展
    def default(self, obj):
        if isinstance(obj, datetime.datetime):
            return obj.strftime('%Y-%m-%d %H:%M:%S')
        elif isinstance(obj, date):
            return obj.strftime('%Y-%m-%d')
        return json.JSONEncoder.default(self, obj)
d=datetime.datetime.now()
print json.dumps(d, cls = DateTimeEncoder)      #
使用cls
指定编码器的名称
```

最后需要提醒的是，Python中标准模块json的性能比pickle与cPickle稍逊。如果对序列化性能要求非常高的场景，可以选择cPickle模块。图4-5显示的是这三者序列化时随着数据规模增加所消耗时间改变的图例。

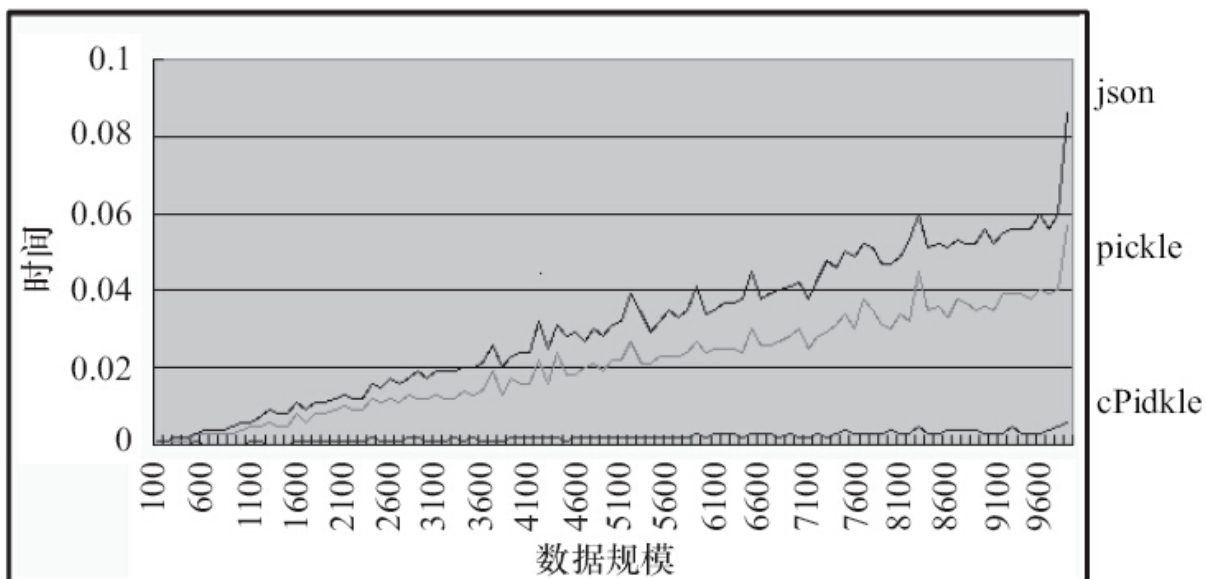


图4-5 pickle、json、cPickle序列化文件时性能比较

建议46：使用traceback获取栈信息

当程序产生异常的时候，最需要面对异常的其实是开发人员，他们需要更多的异常提示的信息，以便调试程序中潜在的错误和问题。先来看一个简单的例子：

```
gList = ['a','b','c','d','e','f','g']
def f():
    gList[5]
    return g()
def g():
    return h()
def h():
    del gList[2]
    return i()
def i():
    gList.append('i')
    print gList[7]
if __name__ == '__main__':
    try:
        f()
    except IndexError as ex:
        print "Sorry,Exception occured,you accessed an element out of range"
        print ex
```

上述程序运行输出如下：

```
Sorry,Exception occured,you accessed an element out of range
list index out of range
```

信息提示有异常产生，对于用户这点还算是较为友好的，那么对于开发人员，他如何快速地知道错误发生在哪里呢？逐行检查代码吗？对于简单的程序，这也不失为一个办法，但在较为复杂的应用程序中这个方法就显得有点低效了。面对异常开发人员最希望看到的往往是异常发生时候的现场信息，**traceback**模块可以满足这个需求，它会输出完整的栈信息。将上述代码异常部分修改如下：

```
except IndexError as ex:
    print "Sorry,Exception occured,you accessed an element out of range"
    print ex
    traceback.print_exc()
```

程序运行会输出异常发生时候完整的栈信息，包括调用顺序、异常发生的语句、错误类型等。

```
Sorry,Exception occured,you accessed an element out of rangelist index out of
range
Traceback (most recent call last):
  File "trace.py", line 20, in <module>
    f()
  File "trace.py", line 5, in f
    return g()
  File "trace.py", line 8, in g
    return h()
  File "trace.py", line 12, in h
    return i()
  File "trace.py", line 16, in i
    print gList[7]
IndexError: list index out of range
```

`traceback.print_exc()`方法打印出的信息包括3部分：错误类型（`IndexError`）、错误对应的值（`list index out of range`）以及具体的`trace`信息，包括文件名、具体的行号、函数名以及对应的源代码。`Traceback`模块提供了一系列方法来获取和显示异常发生时候的`trace`相关信息，下面列举几个常用的方法：

1) `traceback.print_exception(type, value, traceback[, limit[, file]])`，根据`limit`的设置打印栈信息，`file`为`None`的情况下定位到`sys.stderr`，否则则写入文件；其中`type`、`value`、`traceback`这3个参数对应的值可以从`sys.exc_info()`中获取。

2) `traceback.print_exc([limit[, file]])`，为`print_exception()`函数的缩写，不需要传入`type`、`value`、`traceback`这3个参数。

3) `traceback.format_exc([limit])`，与`print_exc()`类似，区别在于返回形式为字符串。

4) `traceback.extract_stack([file[, limit]])`，从当前栈帧中提取trace信息。

读者可以参看Python文档获取更多关于traceback所提供的抽取、格式化或者打印程序运行时候的栈跟踪信息的方法。本质上模块traceback获取异常相关的数据都是通过`sys.exc_info()`函数得到的。当有异常发生的时候，该函数以元组的形式返回(`type`, `value`, `traceback`)，其中`type`为异常的类型，`value`为异常本身，`traceback`为异常发生时候的调用和堆栈信息，它是一个traceback对象，对象中包含出错的行数、位置等数据。上面的例子中也可以通过如下方式输出异常发生时候的详细信息：

```
tb_type, tb_val, exc_tb = sys.exc_info()
for filename, lineno, funcname, source in traceback.extract_tb(exc_tb):
    print "%-23s:%s '%s' in %s()" % (filename, lineno, source, funcname)
```

实际上除了traceback模块本身，inspect模块也提供了获取traceback对象的接口，`inspect.trace([context])`可以返回当前帧对象以及异常发生时进行捕获的帧对象之间的所有栈帧记录列表，因此第一个记录代表当前调用对象，最后一个代表异常发生时候的对象。其中每一个列表元素都是一个由6个元素组成的元组：（frame对象，文件名，当前行号，函数名，源代码列表，当前行在源代码列表中的位置）。本节开头的例子在异常部分使用`inspect.trace()`来获取异常发生时候的堆栈信息，其部分输出结果如下：

```
[(<frame object at 0x022CB480>,
  'testinspect.py',
  23,
  '<module>',
  ['\t\t\t\t\tf()\n'],
  0),
 ... ...]
```

此外如果想进一步追踪函数调用的情况，还可以通过inspect模块的inspect.stack()函数查看函数层级调用的栈相信信息。因此，当异常发生的时候，合理使用上述模块中的方法可以快速地定位程序中的问题所在。

建议47：使用logging记录日志信息

仅仅将栈信息输出到控制台是远远不够的，更为常见的是使用日志保存程序运行过程中的相关信息，如运行时间、描述信息以及错误或者异常发生时候的特定上下文信息。Python中自带的logging模块提供了日志功能，它将logger的level分为5个级别（如表4-3所示），可以通过Logger.setLevel(lvl)来设置，其中DEBUG为最低级别，CRITICAL为最高级别，默认的级别为WARNING。

表4-3 日志级别

Level	使用情形
DEBUG	详细的信息，在追踪问题的时候使用
INFO	正常的信息
WARNING	一些不可预见的问题发生，或者将要发生，如磁盘空间低等，但不影响程序的运行
ERROR	由于某些严重的问题，程序中的一些功能受到影响
CRITICAL	严重的错误，或者程序本身不能够继续运行

logging lib包含以下4个主要对象：

1) **logger**。logger是程序信息输出的接口，它分散在不同的代码中，使得程序可以在运行的时候记录相应的信息，并根据设置的日志级别或filter来决定哪些信息需要输出，并将这些信息分发到其关联的handler。常用的方法有Logger.setLevel()、Logger.addHandler()、Logger.removeHandler()、Logger.addFilter()、Logger.debug()、Logger.info()、Logger.warning()、Logger.error()、etLogger()等。

2) **Handler**。Handler用来处理信息的输出，可以将信息输出到控制台、文件或者网络。可以通过Logger.addHandler()来给logger对象添加handler，常用的handler有StreamHandler和FileHandler类。StreamHandler发送错误信息到流，而FileHandler类用于向文件输出日志

信息，这两个handler定义在logging的核心模块中。其他的handler定义在logging.handlers模块中，如HTTPHandler、SocketHandler。

3) **Formatter**。决定log信息的格式，格式使用类似于%(dictionary key)s的形式来定义，如'%(asctime)s - %(levelname)s - %(message)s'，支持的key可以在Python自带的文档LogRecord attributes中查看。

4) **Filter**。用来决定哪些信息需要输出。可以被handler和logger使用，支持层次关系，比如，如果设置了filter名称为A.B的logger，则该logger和其子logger的信息会被输出，如A.B、A.B.C。

logging.basicConfig(**kwargs))提供对日志系统的基本配置，默认使用StreamHandler和Formatter并添加到root logger，该方法自Python2.4开始可以接受字典参数，支持的字典参数如表4-4所示。

表4-4 字典参数格式类型

格 式	描 述
filename	指定 FileHandler 的文件名，而不是默认的 StreamHandler
filemode	打开文件的模式，同 open 函数中的同名参数，默认为' a '
format	输出格式字符串
datefmt	日期格式
level	设置根 logger 的日志级别
stream	指定 StreamHandler。这个参数若与 filename 冲突，忽略 stream

我们通过修改上一节的例子来看如何结合traceback和logging，记录程序运行过程中的异常。

```
import traceback
import sys
import logging
glList = ['a','b','c','d','e','f','g']
logging.basicConfig( #
配置日志的输出方式及格式
    level=logging.DEBUG,
    filename='log.txt',
    filemode='w',
```

```

    format='%(%asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s',
)
def f():
    gList[5]
    logging.info('[INFO]:calling method g() in f()')#
记录正常的信息
    return g()
def g():
    logging.info('[INFO]:calling method h() in g()')
    return h()
def h():
    logging.info('[INFO]:Delete element in gList in h()')
    del gList[2]
    logging.info('[INFO]:calling method i() in h()')
    return i()
def i():
    logging.info('[INFO]:Append element i to gList in i()')
    gList.append('i')
    print gList[7]
if __name__ == '__main__':
    logging.debug('Information during calling f():')
    try:
        f()
    except IndexError as ex:
        print "Sorry,Exception occured,you accessed an element out of range"
        #traceback.print_exc()
        ty,tv,tb = sys.exc_info()
        logging.error("[ERROR]:Sorry,Exception occured,you accessed an
        element out of range")#
记录异常错误信息
        logging.critical('object info:%s' %ex)
        logging.critical('Error Type:{0},Error Information:{1}'.format(ty,
tv))
        #
记录异常的类型和对应的值
        logging.critical(''.join(traceback.format_tb(tb)))#
记录具体的trace
信息
        sys.exit(1)

```

修改程序后在控制台上对用户仅显示错误提示信息“Sorry,Exception occured,you accessed an element out of range”，而开发人员如果需要debug可以在日志文件中找到具体运行过程中的信息。

```

#
为了节省篇幅仅显示部分日志
2013-06-26 12:05:18,923 traceexample.py[line:41] CRITICAL object info:list
index out of range
2013-06-26 12:05:18,923 traceexample.py[line:42] CRITICAL Error Type:<type
'exceptions.IndexError'>,Error Information:list index out of range
2013-06-26 12:05:18,924 traceexample.py[line:43] CRITICAL File
"traceexample.py",
line 35, in <module>
f()
File "traceexample.py", line 15, in f
return g()
File "traceexample.py", line 19, in g
return h()
File "traceexample.py", line 25, in h

```

```
return i()
File "traceexample.py", line 30, in i
print gList[7]
```

上面的代码中控制运行输出到console上用的是print(), 但这种方法比较原始, logging模块提供了能够同时控制输出到console和文件的方法。下面的例子中通过添加StreamHandler并设置日志级别为logging.ERROR, 可以在控制台上输出错误信息。

```
console = logging.StreamHandler()
console.setLevel(logging.ERROR)
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
console.setFormatter(formatter)
logging.getLogger('').addHandler(console)
```

为了使Logging使用更为简单可控, logging支持loggin.config进行配置, 支持dictConfig和fileConfig两种形式, 其中fileConfig是基于configparser()函数进行解析, 必须包含的内容为[loggers]、[handlers]和[formatters]。具体例子示意如下:

```
[loggers]
keys=root
[logger_root]
level=DEBUG
handlers=hand01
[handlers]
keys=hand01
[handler_hand01]
class=StreamHandler
level=INFO
formatter=form01
args=(sys.stderr,)
[formatters]
keys=form01
[formatter_form01]
format=%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s
datefmt=%a, %d %b %Y %H:%M:%S
```

最后关于logging的使用, 提以下几点建议:

1) 尽量为logging取一个名字而不是采用默认, 这样当在不同的模块中使用的时候, 其他模块只需要使用以下代码就可以方便地使用同

一个logger，因为它本质上符合单例模式。

```
import logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger( __name__ )
```

2) 为了方便地找出问题所在，logging的名字建议以模块或者class来命名。Logging名称遵循按“.”划分的继承规则，根是root logger，logger a.b的父logger对象为a。

3) Logging只是线程安全的，不支持多进程写入同一个日志文件，因此对于多个进程，需要配置不同的日志文件。

建议48：使用threading模块编写多线程程序

GIL的存在使得Python多线程编程暂时无法充分利用多处理器的优势，这种限制也许使很多人感到沮丧，但事实上这并不意味着我们需要放弃多线程。的确，对于只含纯Python的代码也许使用多线程并不能提高运行速率，但在以下几种情况，如等待外部资源返回，或者为了提高用户体验而建立反应灵活的用户界面，或者多用户应用程序中，多线程仍然是一个比较好的解决方案。Python为多线程编程提供了两个非常简单明了的模块：`thread`和`threading`。那么，这两个模块在多线程处理上有什么区别呢？简单一点说：`thread`模块提供了多线程底层支持模块，以低级原始的方式来处理和控制线程，使用起来较为复杂；而`threading`模块基于`thread`进行包装，将线程的操作对象化，在语言层面提供了丰富的特性。Python多线程支持用两种方式来创建线程：一种是通过继承`Thread`类，重写它的`run()`方法（注意，不是`start()`方法）；另一种是创建一个`threading.Thread`对象，在它的初始化函数（`__init__()`）中将可调用对象作为参数传入。实际应用中，推荐优先使用`threading`模块而不是`thread`模块，（除非有特殊需要）。下面来具体分析一下这么做的原因。

1) `threading`模块对同步原语的支持更为完善和丰富。就线程的同步和互斥来说，`thread`模块只提供了一种锁类型`thread.LockType`，而`threading`模块中不仅有`Lock`指令锁、`RLock`可重入指令锁，还支持条件变量`Condition`、信号量`Semaphore`、`BoundedSemaphore`以及`Event`事件等。

2) `threading`模块在主线程和子线程交互上更为友好，`threading`中的`join()`方法能够阻塞当前上下文环境的线程，直到调用此方法的线程终止或到达指定的`timeout`（可选参数）。利用该方法可以方便地控制主线程和子线程以及子线程之间的执行。来看一个简单示例：

```
import threading, time, sys
class test(threading.Thread):
    def __init__(self, name, delay):
        threading.Thread.__init__(self)
        self.name = name
        self.delay = delay
    def run(self):
        print "%s delay for %s" %(self.name, self.delay)
        time.sleep(self.delay)
        c = 0
        while True:
            print "This is thread %s on line %s" %(self.name, c)
            c = c + 1
            if c == 3:
                print "End of thread %s" % self.name
                break
t1 = test('Thread 1', 2)
t2 = test('Thread 2', 2)
t1.start()
print "Wait t1 to end"
t1.join()
t2.start()
print 'End of main'
```

上面的例子中，主线程`main`在`t1`上使用`join()`的方法，主线程会等待`t1`结束后才继续运行后面的语句，由于线程`t2`的启动在`join`语句之后，`t2`一直等到`t1`退出后才会开始运行。输出结果如图4-6所示。

```

Thread 1 delay for 2Wait t1 to end

This is thread Thread 1 on line 0
This is thread Thread 1 on line 1
This is thread Thread 1 on line 2
End of thread Thread 1
Thread 2 delay for 2End of main

This is thread Thread 2 on line 0
This is thread Thread 2 on line 1
This is thread Thread 2 on line 2
End of thread Thread 2

```

图4-6 多线程示例输出结果

3) `thread`模块不支持守护线程。`thread`模块中主线程退出的时候,所有的子线程不论是否还在工作,都会被强制结束,并且没有任何警告也没有任何退出前的清理工作。来看一个例子:

```

from thread import start_new_thread
import time
def myfunc(a,delay):
    print "I will calculate square of  %s after delay for %s" %(a,delay)
    time.sleep(delay)
    print "calculate begins..."
    result = a*a
    print result
    return result
start_new_thread(myfunc,(2,5))#
同时启动两个线程
start_new_thread(myfunc,(6,8))
time.sleep(1)

```

运行程序,输出结果如下,你会发现子线程的结果还未返回就已经结束了。

```

I will calculate square of  2 after delay for 5I will calculate square of  6
after delay for 2

```

这是一种非常野蛮的主线程和子线程的交互方式。如果把主线程和子线程组成的线程组比作一个团队的话，那么主线程应该是这个团队的管理者，它了解每个线程所做的事情、所需的数据输入以及子线程结束时的输出，并把各个线程的输出组合形成有意义的结果。如果一个团队中管理者采取这种强硬的管理方式，相信很多下属都会苦不堪言，因为不仅没有被尊重的感觉，而且还有可能因为这种强势带来决策上的失误。实际上很多情况下我们可能希望主线程能够等待所有子线程都完成时才退出，这时应该使用threading模块，它支持守护线程，可以通过setDaemon()函数来设定线程的daemon属性。当daemon属性设置为True的时候表明主线程的退出可以不用等待子线程完成。默认情况下，daemon标志为False，所有的非守护线程结束后主线程才会结束。来看具体的例子，当daemon属性设置为False，默认主线程会等待所有子线程结束才会退出。将t2的daemon属性改为True之后即使t2运行未结束主线程也会直接退出。

```
import threading
import time
def myfunc(a,delay):
    print "I will calculate square of  %s after delay for %s" %(a,delay)
    time.sleep(delay)
    print "calculate begins..."
    result = a*a
    print result
    return result
t1=threading.Thread(target=myfunc,args=(2,5))
t2=threading.Thread(target=myfunc,args=(6,8))
print t1.isDaemon()
print t2.isDaemon()
t2.setDaemon(True) #
设置守护线程
t1.start()
t2.start()
```

4) Python3中已经不存在thread模块。thread模块在Python3中被命名为_thread，这种更改主要是为了更进一步明确表示与thread模块相关的更多的是具体的实现细节，它更多展示的是操作系统层面的原始操作和处理。在一般的代码中不应该选择thread模块。

建议49：使用Queue使多线程编程更安全

曾经有这么一个说法，程序中存在3种类型的bug：你的bug、我的bug和多线程。这虽然是句调侃，但从某种程度上道出了一个事实：多线程编程不是件容易的事情。线程间的同步和互斥，线程间数据的共享等这些都是涉及线程安全要考虑的问题。纵然Python中提供了众多的同步和互斥机制，如mutex、condition、event等，但同步和互斥本身就不是一个容易的话题，稍有不慎就会陷入死锁状态或者威胁线程安全。我们来看一个经典的多线程同步问题：生产者消费者模型。如果用Python来实现，你会怎么写？大概思路是这样的：分别创建消费者和生产者线程，生产者往队列里面放产品，消费者从队列里面取出产品，创建一个线程锁以保证线程间操作的互斥性。当队列满的时候消费者进入等待状态，当队列空的时候生产者进入等待状态。我们来看一个具体的Python实现：

```
import Queue
import threading
import random
writelock = threading.Lock()                                #
创建锁对象用于控制输出
class Producer(threading.Thread):
    def __init__(self, q, con, name):
        super(Producer, self).__init__()
        self.q = q
        self.name = name
        self.con = con
        print "Producer "+self.name+" Started"
    def run(self):
        while 1:
            global writelock
            self.con.acquire()                                #
获取锁对象
            if self.q.full():                                  #
队列满
                with writelock:                                #
                    print 'Queue is full,producer wait!'
                    self.con.wait()                            #
                    等待资源
            else:
                value = random.randint(0,10)
                with writelock:
```

```

                                print self.name + " put value"
                                "+self.name+": "+ str(value)+
                                "into queue"
放入队列中                    self.q.put((self.name+": "+str(value)))          #
                                self.con.notify()                            #
通知消费者                    self.con.release()                            #
释放锁对象
class Consumer(threading.Thread):    #
消费者
    def __init__(self, q, con, name):
        super(Consumer, self).__init__()
        self.q = q
        self.con = con
        self.name = name
        print "Consumer "+self.name+" started\n "
    def run(self):
        while 1:
            global writelock
            self.con.acquire()
            if self.q.empty():          #
                with writelock:
                    print 'queue is empty, consumer
wait!'
                self.con.wait()          #
            else:
                value = self.q.get()      #
            with writelock:
                print self.name + "get value"+
                value + " from queue"
            self.con.notify()            #
            self.con.release()            #
            发送消息通知生产者
        释放锁对象
if __name__ == "__main__":
    q = Queue.Queue(10)
    con = threading.Condition()        #
    条件变量锁
    p = Producer(q, con, "P1")
    p.start()
    p1 = Producer(q, con, "P2")
    p1.start()
    c1 = Consumer(q, con, "C1")
    c1.start()

```

上面的程序实现有什么问题吗？回答这个问题之前，我们先来了解一下Queue模块的基本知识。Python中的Queue模块提供了3种队列：

- **Queue.Queue(maxsize)**：先进先出，maxsize为队列大小，其值为非正数的时候为无限循环队列。

·`Queue.LifoQueue(maxsize)`：后进先出，相当于栈。

·`Queue.PriorityQueue(maxsize)`：优先级队列。

·这3种队列支持以下方法：

·`Queue.qsize()`：返回近似的队列大小。注意，这里之所以加“近似”二字，是因为当该值 >0 的时候并不保证并发执行的时候`get()`方法不被阻塞，同样，对于`put()`方法有效。

·`Queue.empty()`：列队为空的时候返回`True`，否则返回`False`。

·`Queue.full()`：当设定了队列大小的情况下，如果队列满则返回`True`，否则返回`False`。

·`Queue.put(item[, block[, timeout]])`：往队列中添加元素`item`，`block`设置为`False`的时候，如果队列满则抛出`Full`异常。如果`block`设置为`True`，`timeout`为`None`的时候则会一直等待直到有空位置，否则会根据`timeout`的设定超时后抛出`Full`异常。

·`Queue.put_nowait(item)`：等价于`put(item, False)`。`block`设置为`False`的时候，如果队列空则抛出`Empty`异常。如果`block`设置为`True`、`timeout`为`None`的时候则会一直等待直到有元素可用，否则会根据`timeout`的设定超时后抛出`Empty`异常。

·`Queue.get([block[, timeout]])`：从队列中删除元素并返回该元素的值。

·`Queue.get_nowait()`：等价于`get(False)`。

·`Queue.task_done()`：发送信号表明入列任务已经完成，经常在消费者线程中用到。

·`Queue.join()`：阻塞直至队列中所有的元素处理完毕。

`Queue`模块实现了多个生产者多个消费者的队列，当多线程之间需要信息安全的交换的时候特别有用，因此这个模块实现了所需要的锁原语，为Python多线程编程提供了有力的支持，它是线程安全的。需要注意的是`Queue`模块中的列队和`collections.deque`所表示的队列并不一样，前者主要用于不同线程之间的通信，它内部实现了线程的锁机制；而后者主要是数据结构上的概念，因此支持`in`方法。

再回过头来看看前面的例子，程序的实现有什么问题呢？答案很明显，作用于`queue`操作的条件变量完全是不需要的，因为`queue`本身能够保证线程安全，因此不需要额外的同步机制。那么，该如何修改呢？请读者自行思考。下面的多线程下载的例子也许有助于你完成上面程序的修改。

```
import os
import Queue
import threading
import urllib2
class DownloadThread(threading.Thread):
    def __init__(self, queue):
        threading.Thread.__init__(self)
        self.queue = queue
    def run(self):
        while True:
            url = self.queue.get()          #
            从队列中取出一个url元素
            print self.name+"begin download"+url+"..."
            self.download_file(url)         #
            进行文件下载
            self.queue.task_done()          #
            下载完毕发送信号
            print self.name+" download completed!!!"
        def download_file(self, url):       #
            下载文件
            urlhandler = urllib2.urlopen(url)
            fname = os.path.basename(url)+".html" #
            文件名称
            with open(fname, "wb") as f:    #
            打开文件
                while True:
                    chunk = urlhandler.read(1024)
                    if not chunk: break
                    f.write(chunk)
if __name__ == "__main__":
    urls = ["http://wiki.python.org/moin/WebProgramming",
```

```
        "https://www.createspace.com/3611970",
        "http://wiki.python.org/moin/Documentation"
    ]
    queue = Queue.Queue()
    # create a thread pool and give them a queue
    for i in range(5):
        t = DownloadThread(queue)          #
        # 启动5个线程同时进行下载
        t.setDaemon(True)
        t.start()
    # give the queue some data
    for url in urls:
        queue.put(url)
    # wait for the queue to finish
    queue.join()
```

第5章 设计模式

设计模式由来已久，并广泛存在于各行各业中，不过软件开发行业的设计模式广为人知，这还是GoF的《设计模式——可复用面向对象软件的基础》一书之功，而后来的《Head First设计模式》则通过幽默有趣的文风使其广泛流行于程序员之间。这两本书堪称经典，但是它们分别使用C++和Java编程语言作为载体，如果照搬到Python程序中，代码里散发出的浓浓的静态语言风格让人无所适从。笔者坚信在使用Python语言进行编程时得当地应用设计模式是有益的，而且Python的动态语言特性并不能完全替代设计模式。所以本章的内容主要聚焦于如何编写Pythonic的设计模式代码，让设计模式这一高层思想更好地落实到我们的编程实践中去。

建议50： 利用模块实现单例模式

在GOF的23种设计模式中，单例是最常使用的模式，通过单例模式可以保证系统中一个类只有一个实例而且该实例易于被外界访问，从而方便对实例个数的控制并节约系统资源。每当大家想要实现一个名为**XxxManger**的类时，往往意味着这是一个单例。在游戏编程中尤其是如此，比如一个名为**World**的单例管理着游戏中的所有资源，包括一个名为**Sun**的单例，它给这个世界带来了光亮。

单例如此常见，所以有不少现代编程语言将其加到了语言特性中，如**scala**和**falcon**语言都把**object**定义成关键词，并用其声明单例。如在**scala**中，一个单例如下：

```
object Singleton {  
  def show = println("I am a singleton")  
}
```

object定义了一个名为**Singleton**的单例，它满足单例的3个需求：一是只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。对于第三点，在任何地方都可以通过调用**Singleton.show()**来验证。在**scala**中，单例没有显式的初始化操作，但并不是所有在语法层面支持单例模式的编程语言都如此，比如**falcon**就不一样。

```
object object_name [ from class1, class2 ... classN]  
  property_1 = expression  
  property_2 = expression  
  ...  
  property_N = expression  
  [init block]  
  function method_1( [parameter_list] )  
    [method_body]  
  end  
  ...
```

```
function method_N( [parameter_list] )
  [method_body]
end
end
```

上面是falcon语言的单例语法，[init block]能够让程序员手动控制单例的初始化代码。但是与scala和falcon相比，动态语言Python就没有那么方便了，主要原因是Python缺乏声明私有构造函数的语法元素，实例又带有类型信息。比如以下方法是不可行的：

```
class _Singleton(object):
    pass
Singleton = _Singleton()
del _Singleton                                #
# 试图删除 class
# 定义
another = Singleton.__class__()              #
# 没用，绕过！
print(type(another))
#
# 输出
<class '__main__._Singleton'>
```

可见虽然把Singleton的类定义删除了，但仍然有办法通过已有实例的__class__属性生成一个新的实例。于是许多Pythonista把目光聚集到真正创建实例的方法__new__上，并做起了文章。

```
class Singleton(object):
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(
                cls, *args, **kwargs)
        return cls._instance
if __name__ == '__main__':
    s1=Singleton()
    s2=Singleton()
    assert id(s1)==id(s2)
```

这个方法很好地解决了前面的问题，现在基本上可以保证“只能有一个实例”的要求了，但是在并发情况下可能会发生意。为了解决这个问题，引入一个带锁的版本。

```
class Singleton(object):
    objs = {}
    objs_locker = threading.Lock()
    def __new__(cls, *args, **kv):
        if cls in cls.objs:
            return cls.objs[cls]
        cls.objs_locker.acquire()
        try:
            if cls in cls.objs: ## double check locking
                return cls.objs[cls]
            cls.objs[cls] = object.__new__(cls)
        finally:
            cls.objs_locker.release()
```

利用经典的双检查锁机制，确保了在并发环境下Singleton的正确实现。但这个方案并不完美，至少还有以下两个问题：

- 如果Singleton的子类重载了__new__()方法，会覆盖或者干扰Singleton类中__new__()的执行，虽然这种情况出现的概率极小，但不可忽视。

- 如果子类有__init__()方法，那么每次实例化该Singleton的时候，__init__()都会被调用到，这显然是不应该的，__init__()只应该在创建实例的时候被调用一次。

这两个问题当然可以解决，比如通过文档告知其他程序员，子类化Singleton的时候，请务必记得调用父类的__new__()方法；而第二个问题也可以通过偷偷地替换掉__init__()方法来确保它只调用一次。但是，为了实现一个单例，做大量的、水面之下的工作让人感觉相当不Pythonic。这也引起了Python社区的反思，有人开始重新审视Python的语法元素，发现模块采用的其实是天然的单例的实现方式。

- 所有的变量都会绑定到模块。

- 模块只初始化一次。

·import机制是线程安全的（保证了在并发状态下模块也只有一个实例）。

当我们想要实现一个游戏世界时，只需简单地创建World.py就可以了。

```
# World.py
import Sun
def run():
    while True:
        Sun.rise()
        Sun.set()
```

然后在入口文件main.py里导入，并调用run()函数，看，是不是感觉这种方式最为Pythonic呢？

```
# main.py
import World
World.run()
```



注意

Alex Martelli认为单例模式要求“实例的唯一性”本身是有问题的，实际更值得关注的是实例的状态，只要所有的实例共享状态（可以狭义地理解为属性）、行为（可以狭义地理解为方法）一致就可以了。在这一思想的进一步指导下，他提出了Borg模式（在C#中又称为Monostate模式）。

```
class Borg:
    __shared_state = {}
    def __init__(self):
        self.__dict__ = self.__shared_state
    # and whatever else you want in your class -- that's all!
```

通过**Borg**模式，可以创建任意数量的实例，但因为它们共享状态，从而保证了行为一致。虽然Alex的这个**Borg**模式仅适用于古典类（**classic class**），Python 2.2版本以后的新式类（**new-style classes**）需要使用__getattr__和__setattr__方法来实现（代码略），但其可开阔眼界。

建议51：用mixin模式让程序更加灵活

在理解mixin之前，有必要先重温一下模板方法模式。所谓的模板方法模式就是在一个方法中定义一个算法的骨架，并将一些实现步骤延迟到子类中。模板方法可以使子类在不改变算法结构的情况下，重新定义算法中的某些步骤。在这里，算法也可以理解为行为。

模板方法模式在C++或其他语言中并无不妥，但是在Python语言中，则颇有点画蛇添足的味道。比如模板方法，需要先定义一个基类，而实现行为的某些步骤则必须在其子类中，在Python中并无必要。

```
class People(object):
    def make_tea(self):
        teapot = self.get_teapot()
        teapot.put_in_tea()
        teapot.put_in_water()
        return teapot
```

在这个例子中，`get_teapot()`方法并不需要预先定义。假设在上班时，使用的是简易茶壶，而在家里，使用的是功夫茶壶，那么可以这样编写代码：

```
class OfficePeople(People):
    def get_teapot(self):
        return SimpleTeapot()
class HomePeople(People):
    def get_teapot(self):
        return KungfuTeapot()
```

这段代码工作得很好，虽然看起来像模板方法，但是基类并不需要预先声明抽象方法，甚至还带来调试代码的便利。假定存在一个People的子类StreetPeople，用以描述“正走在街上的人”，作为“没有人

会随身携带茶壶”的常识的反映，这个类将不会实现`get_teapot()`方法，所以一调用`make_tea()`就会产生一个找不到`get_teapot()`方法的`AttributeError`。由此程序员马上会想到“正走在街上的人”边走边泡茶这样的需求是不合理的，从而能够在更高层次上考虑业务的合理性，在更接近本源的地方修正错误。

但是，这段代码并不完美。老板（`OfficePeople`的一个实例）拥有巨大的办公室，他购置了功夫茶具，他要在办公室喝功夫茶了。怎么办？答案有两种，一种是从`OfficePeople`继承子类`Boss`，重写它的`get_teapot()`，使它返回功夫茶具；另一个则是把`get_teapot()`方法提取出来，把它以多继承的方式做一次静态混入。

```
class UseSimpleTeapot(object):
    def get_teapot(self):
        return SimpleTeapot()
class UseKungfuTeapot(object):
    def get_teapot(self):
        return KungfuTeapot()
class OfficePeople(People, UseSimpleTeapot):pass
class HomePeople(People, UseKungfuTeapot):pass
class Boss(People, UseKungfuTeapot):pass
```

这样就很好地解决了老板在办公室也要喝功夫茶的需求。但是这样的代码仍然没有把Python的动态性表现出来：当新的需求出现时，需要更改类定义。比如随着公司扩张，越来越多的人入职，`OfficePeople`的需求越来越多，开始出现有人不喝茶而是喝咖啡，也有人既喜欢喝茶还喜欢喝咖啡，出现了喜欢在独立办公室抽雪茄的职业经理人……这些类越来越多，代码越发难以维护。让我们开始寄望于动态地生成不同的实例。

```
def simple_tea_people():
    people = People()
    people.__bases__ += (UseSimpleTeapot,)
    return people
def coffee_people():
    people = People()
    people.__bases__ += (UseCoffeePot,)
    return people
```

```
def tea_and_coffee_people():
    people = People()
    people.__bases__ += (UseSimpleTeapot, UseCoffeepot,)
    return people
def boss():
    people = People()
    people.__bases__ += (KungfuTeapot, UseCoffeepot,)
    return people
```

这个代码能够运行的原理是，每个类都有一个__bases__属性，它是一个元组，用来存放所有的基类。与其他静态语言不同，Python语言中的基类在运行中可以动态改变。所以当我们向其中增加新的基类时，这个类就拥有了新的方法，也就是所谓的混入（mixin）。这种动态性的好处在于代码获得了更丰富的扩展功能。想象一下，你之前写好的代码并不需要个性，只要后期为它增加基类，就能够增强功能

（或替换原有行为），这多么方便！值得进一步探索的是，利用反射技术，甚至不需要修改代码。假定我们在OA系统里定义员工的时候，有一个特性选择页面，在里面可以勾选该员工的需求。比如对于Boss，可以勾选功夫茶和咖啡，那么通过的代码可能如下：

```
import mixins
def staff():
    people = People()
    bases = []
    for i in config.checked():
        bases.append(getattr(mixins, i))
    people.__bases__ += tuple(bases)
    return people
```

看，通过这个框架代码，OA系统的开发人员只需要把员工常见的需求定义成Mixin预告放在mixins模块中，就可以在不修改代码的情况下通过管理界面满足几乎所有需求了。Python的动态性优势也在这个例子中得到了很好的展现。

建议52：用发布订阅模式实现松耦合

发布订阅模式（publish/subscribe或pub/sub）是一种编程模式，消息的发送者（发布者）不会发送其消息给特定的接收者（订阅者），而是将发布的消息分为不同的类别直接发布，并不关注订阅者是谁。而订阅者可以对一个或多个类别感兴趣，且只接收感兴趣的消息，并且不关注是哪个发布者发布的消息。这种发布者和订阅者的解耦可以允许更好的可扩充性和更为动态的网络拓扑，故受到了大家的喜爱。

发布订阅模式的优点是发布者与订阅者松散的耦合，双方不需要知道对方的存在。由于主题是被关注的，发布者和订阅者可以对系统拓扑毫无所知。无论对方是否存在，发送者和订阅者都可以继续正常操作。要实现这个模式，就需要有一个中间代理人，在实现中一般被称为**Broker**，它维护着发布者和订阅者的关系：订阅者把感兴趣的主题告诉它，而发布者的信息也通过它路由到各个订阅者处。简单的实现如下：

```
from collections import defaultdict
route_table = defaultdict(list)
def sub(self, topic, callback):
    if callback in route_table[topic]:
        return
    route_table[topic].append(callback)
def pub(self, topic, *a, **kw):
    for func in route_table[topic]:
        func(*a, **kw)
```

这个实现非常简单，直接放在一个叫**Broker.py**的模块中（这显然是单件），省去了各种参数检测、优先处理的需求等，甚至没有取消订阅的函数，但它的确展现了发布订阅模式实现的最基础的结构，它的应用代码也可以运行。

```
import Broker
def greeting(name):
    print 'Hello, %s.' % name
Broker.sub('greet', greeting)
Broker.pub('greet', 'LaiYonghao')
#
输出
Hello, LiaYonghao.
```

相对于这个简化版本，**blinker**和**python-message**两个模块的实现要完备得多。**blinker**已经被用在了多个广受欢迎的项目上，比如**flask**和**django**；而**python-message**则支持更多丰富的特性。本节以**python-message**的使用为例，讲解发布订阅模式的应用场景。

安装**python-message**相当简单，通过**pip**安装就可以了。

```
pip install message
```

然后简单验证一下。

```
import message
def hello(name):
    print 'hello, %s.' % name
message.sub('greet', hello)
message.pub('greet', 'lai')
```

运行输出如下：

```
hello, lai.
```

接下来用它解决一些实际问题。假定你给项目组开发了一个程序库**foo**，里面有一个非常重要的函数——**bar**。

```
def bar():
    print 'Haha, Calling bar().'
    do_sth()
```

这个函数如此重要，所以你给它加上了一行输出代码，用以输出日志。后来你的这个程序库foo被大量使用了，一直运行得很好，直到又一个新项目拖你过去“救火”，因为出了bug无法查出原因，怀疑是foo的问题。你查看了很久日志，都没有发现他们调用bar()的痕迹，一问，原来他们是用logging的，标准输出在做Daemon的时候被重定向到了/dev/null。在临时修改了输出重定向以后，找到了bug所在，并解决了。然后你开始着手解决这个问题。一开始你想在你的foo库中引入logging，但原来的项目又不用logging，你在程序库里引入logging，但谁来初始化它呢？就算你引入了logging，则你们的项目可能是用logging.getLogger('prjA')获取logger，另一个项目可能是用logging.getLogger('prjB')，日后还有新项目呢！一想到要兼容这么多项目你就头大了。忍痛割爱，把print语句给删除掉吧，又怕日后出了问题自己都找不到bug，那还不是自己加班自己苦。这个时候，不妨让python-message来帮你，轻松改一下bar()函数。

```
import message
LOG_MSG = ('log', 'foo')
def bar():
    message.pub(LOG_MSG, 'Haha, Calling bar().')
    do_sth()
```

在已有的项目中，只需要在项目开始处加上这样的代码，继续把日志放到标准输出。

```
import message
import foo
def handle_foo_log_msg(txt):
    print txt
message.sub(foo.LOG_MSG, handle_foo_log_msg)
```

而在那个使用logging的新项目中，则这样修改：

```
def handle_foo_log_msg(txt):
    import logging
    logging.debug(txt)
```

甚至在一些不关注底层库的日志项目中，直接无视就可以了。通过message，可以轻松获得库与应用之间的解耦，因为库关注的是要有日志，而不关注日志输出到哪里；应用关注的是日志要统一放置，但不关注谁往日志文件中输出内容，这正与发布订阅模式的应用场景不谋而合。

除了简单的sub()/pub()之外，python-message还支持取消订阅（unsub()）和中止消息传递。

```
import message
def hello(name):
    print 'hello %s' % name
    ctx = message.Context()
    ctx.discontinued = True
    return ctx
def hi(name):
    print 'u cann\'t c me.'
message.sub('greet', hello)
message.sub('greet', hi)
message.pub('greet', 'lai')
```

python-message利用回调函数的返回值来实现取消消息传递，非常巧妙（读者可以思考一下为什么能够利用回调函数的返回值）。在上面这个例子中，运行后是看不到“u can't c me.”这一行输出的，因为消息在调用hello()后就中止传递了（Broker使用list对象存储回调函数就是为了保证次序）。

python-message是同步调用回调函数的，也就是说谁先sub谁就先被调用。大部分情况下这样已经能够满足大部分需求，但有时需要后sub的函数先被调用，这时message.sub函数通过一个默认参数来支持的，只需要简单地在调用sub的时候加上front=True，这个回调函数将被插到所有之前已经sub的回调函数之前：sub('greet',hello,front=True)。

订阅/发布模式是观察者模式的超集，它不关注消息是谁发布的，也不关注消息由谁处理。但有时候我们也希望某个自己的类的也能够更方便地订阅/发布消息，也就是想退化为观察者模式，python-message同样提供了支持。如以下代码：

```
from message import observable
def greet(people):
    print 'hello, %s.%s' % people.name
@observable
class Foo(object):
    def __init__(self, name):
        print 'Foo'
        self.name = name
        self.sub('greet', greet)
    def pub_greet(self):
        self.pub('greet', self)
foo = Foo('lai')
foo.pub_greet()
```

python-message提供了类装饰函数observable()，任何class只需要通过它装饰一下就拥有了sub/unsub/pub/declare/retract等方法，它们的使用方法跟全局函数是类似的，在此不赘述。



注意

因为python-message的消息订阅默认是全局性的，所以有可能产生名字冲突。在减少名字冲突方面，可以借鉴java/actionsript3的package起名策略，比如在应用中定义消息主题常量FOO='com.googlecode.python-message.FOO'，这样多个库同时定义FOO常量也不容易冲突。除此之外，还有一招就是使用uuid，如下：

```
uuid = 'bd61825688d72b345ce07057b2555719'
FOO = uuid + 'FOO'
```

建议53：用状态模式美化代码

所谓状态模式，就是当一个对象的内在状态改变时允许改变其行为，但这个对象看起来像是改变了其类。状态模式主要用于控制一个对象状态的条件表达式过于复杂的情况，其可把状态的判断逻辑转移到表示不同状态的一系列类中，进而把复杂的判断逻辑简化。

得益于Python语言的动态性，状态模式的Python实现与C++等语言的版本比起来简单得多。举个例子，一个人，工作日和周日的日常生活是不同的。

```
def workday():
    print 'work hard!'
def weekend():
    print 'play harder!'
class People(object):pass
people = People()
while True:
    for i in xrange(1, 8):
        if i == 6:
            people.day = weekend
        if i == 1:
            people.day = workday
        people.day()
```

运行上述代码，输出如下：

```
work hard!
work hard!
work hard!
work hard!
work hard!
play harder!
play harder!
...
```

就这样，通过在不同的条件下将实例的方法（即行为）替换掉，就实现了状态模式。但是这个简单的例子仍然有以下缺陷：

- 查询对象的当前状态很麻烦。

- 状态切换时需要对原状态做一些清扫工作，而对新的状态需要做一些初始化工作，因为每个状态需要做的事情不同，全部写在切换状态的代码中必然重复，所以需要有一个机制来简化。

python-state包通过几个辅助函数和修饰函数很好地解决了这个问题，并且定义了一个简明状态机框架。先用**pip**安装它。

```
pip install state
```

然后用它改写之前的例子。

```
from state import curr, switch, stateful, State, behavior
@stateful
class People(object):
    class Workday(State):
        default = True
        @behavior
        def day(self):
            print 'work hard.'
    class Weekend(State):
        @behavior
        def day(self):
            print 'play harder!'
people = People()
while True:
    for i in xrange(1, 8):
        if i == 6:
            switch(people, People.Weekend)
        if i == 1:
            switch(people, People.Workday)
    people.day()
```

怎么样？是不是感觉好像比应用模式之前的代码还要长？这是因为例子太简单了，后面再给大家展示更贴近真实业务需求的例子。现在我们先按下这个不表，单看最后一行的**people.day()**。 **people**是**People**的一个实例，但是**People**并没有定义**day()**方法啊？为了解决这个疑惑，需要我们从头看起。

首先是@stateful这个修饰函数，它包含了许多“黑魔法”，其中最重要的是重载了被修饰类的__getattr__()方法从而使得People的实例能够调用当前状态类的方法。被@stateful修饰后的类的实例是带有状态的，能够使用curr()查询当前状态，也可以使用switch()进行状态切换。接下来继续往下看，可以看到类Workday继续自State类，这个State类也是来自于state包，从其派生的子类能够使用__begin__和__end__状态转换协议，通过重载这两个协议，子类能够自定义进入和离开当前状态时对宿主（在本例中即people）的初始化和清理工作。对于一个@stateful类而言，有一个默认的状态（即其实例初始化后的第一个状态），通过类定义的default属性标识，default设置为True的类成为默认状态。@behavior修饰函数用以修饰状态类的方法，其实它是内置函数staticmethod的别名。为什么要将状态类的方法实现为静态方法呢？因为state包的原则是状态类只有行为，没有状态（状态都保存在宿主上），这样可以更好地实现代码重用。那么day()方法既然是静态的，为什么有self参数？这其实是因为self并不是Python的关键字，在这里使用self有助于理解状态类的宿主是People的实例。

至此，读者对state这个简单的包就基本上了解清楚了。下面讲一个来自真实业务的例子，看它如何美化原有的代码。在网络编程中，通常有一个User类，每一个在线用户都有一个User的实例与之对应。User有一些方法，需要确保用户登录之后才能调用，比如查看用户信息。这些方法大概像这样：

```
class User(object):
    def signin(self, usr, pwd):
        ...
        self._signin = True
    def do_sth(self, *a, **kw):
        if not self._signin:
            raise NeedSignin()
        ...
```

真实项目中，类似do_sth()的业务代码数量不少，如果每个函数前两行都是if...raise...，以确保调用场景正确，那么可以想象得出来代码该有多么难看（代码一重复就不好看了）。这时候程序员会选择使用decorator来修饰这些业务代码。

```
def ensure_signin(func):
    def _func(self, *a, **kw):
        if not self._signin:
            raise NeedSignin()
        return func(self, *a, **kw)
    return _func
@ensure_signin
def do_sth(self, *a, **kw):
    ...
```

上述代码看上去很完美的解决方案，而且@ensure_signin相当Pythonic。但是想象一下，某些地方，你除了要确定登录之外，还需要确定是否在战斗副本中，角色是否已经死亡.....等等。想象一下，十个八个方法，每个方法上面都顶着四五个修饰函数，该有多么丑陋！这就是状态模式可以美化的地方。

```
@stateful
class User(object):
    class NeedSignin(State):
        default = True
        @behavior
        def signin(self, usr, pwd):
            ...
            switch(self, Player.Signin)
    class Signin(State):
        @behavior
        def move(self, dst): ...
        @behavior
        def atk(self, other): ...
```

可以看到，当用户登录以后，就切换到了Player.Signin状态，而在Signin状态的行为是不需要做是否已经登录的判断的，这是因为除了登录成功，User的实例无法跳转到Signin状态，反过来说就是只要当前状态是Signin，那必定已经登录，自然无须再验证。

可以看到，通过状态模式，可以像decorator一样去掉if...raise...上下文判断，但比它更棒的是真的一个if...raise...都没有了。另外，需要多重判断的时候要给一个方法戴上四五顶“帽子”的情况也没有了，还通过把多个方法分派到不同的状态类，消灭掉一般情况下Player总是一个巨类的“坏味道”，保持类的短小，更容易维护和重用。不过这些都比不上一个更大的好处：当调用当前状态不存在的行为时，出错信息抛出的是AttributeError，从而避免把问题变为复杂的逻辑错误，让程序员更容易找到出错位置，进而修正问题。

第6章 内部机制

“知其然还必须知其所以然”，除了掌握Python本身的语法以及使用外，对其内部机制的探索可以让我们更深入地理解和掌握语言本身所蕴含的思想和理念。本章将探讨Python的一些内部机制以及高于语法级别的话题，包括名字查找机制、描述符、对象的管理与回收，以及迭代器协议等。通过本章，希望读者可以更好地理解和掌握Python的精髓及其哲学原理。

建议54： 理解built-in objects

我们知道Python中一切皆对象：字符是对象，列表是对象，内建类型（built-in type）也是对象；用户定义的类型是对象，object是对象，type也是对象。自Python2.2之后，为了弥补内建类型和古典类（classic classes）之间的鸿沟 [1] 引入了新式类（new-style classes）。在新式类中，object是所有内建类型的基类，用户所定义的类可以继承自object也可继承自内建类型。

那么内建类型 、object 、type以及用户所定义的类之间到底有什么关系呢？它们之间本质上有什么不同吗？我们来看一个简单的例子：

```
class A:                                     ... ..
①古典类A
    pass
class B(object):
    pass
class C(type):
    pass
class D(dict):                             ... ..
②D
继承自内建类型dict
    pass
```

现在有类A 、B 、C 、D的实例分别对应为a 、b 、c 、d，我们来看一组求值的结果，如表6-1所示。

表6-1 不同对象求值结果

编号	对象	type<*>	*.__class__	*.__bases__	isinstance (*,object)	isinstance (*,type)
[1]	object	<type 'type'>	<type 'type'>	()	真	真
[2]	type	<type 'type'>	<type 'type'>	(<type 'object'>,)	真	真
[3]	a	<type 'instance'>	__main__.A	()	真	假
[4]	b	<class '__main__.B'>	<class '__main__.B'>	(<type 'object'>,)	真	假
[5]	c	<class '__main__.C'>	<class '__main__.C'>	(<type 'type'>,)	真	真
[6]	d	<class '__main__.D'>	<class '__main__.D'>	(<type 'dict'>,)	真	假

注：上表中求值时*用表中第一列的元素代替。

从表6-1中我们可以得出如下结论：

- object[1]和古典类[3]没有基类，type[2]的基类为object。

- 新式类（[4],[5],[6]）中type()的值和__class__的值是一样的，但古典类[3]中实例的type为instance，其type()的值和__class__的值不一样。

- 继承自内建类型的用户类的实例[6]也是object的实例，object[1]是type的实例，type实际是个元类（metaclass）。

- object和内建类型以及所有基于type构建的用户类[5]都是type的实例。

- 在古典类中，所有用户定义的类的类型都为instance。

综上，不同类型的对象之间的关系如图6-1所示。

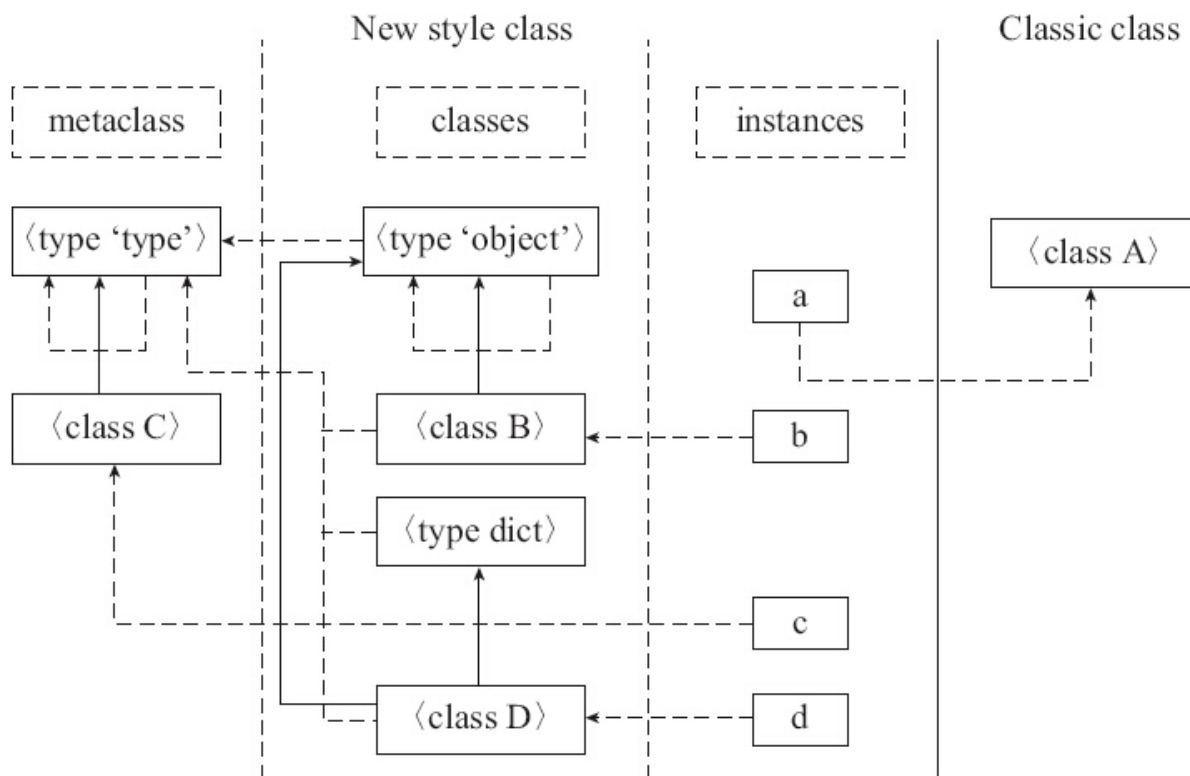


图6-1 不同类型对象的关系图

我们知道古典类和新式类的一个区别是：新式类继承自`object`类或者内建类型。那么是不是可以这么理解：如果一个类定义的时候继承自`object`或者内建类型，那么它就是一个新式类，否则则是古典类？我们来看一个例子：

```
>>> class TestNewClass:
...     __metaclass__ = type      ... .. .
①设置__metaclass__
属性为type
...
>>>
>>> type(TestNewClass)
<type 'type'>
>>> TestNewClass.__bases__
(<type 'object'>,)
>>>
>>> a = TestNewClass()
>>> type(a)
<class '__main__.TestNewClass'>
>>> a.__class__
<class '__main__.TestNewClass'>
>>>
```

从上述例子我们可以看出，`TestNewClass`在定义的时候并没有继承任何类，但测试的结果表明其父类为`object`，它还是属于新式类。这其中的原因在于`TestNewClass`中设置了`__metaclass__`属性（关于元类的更多介绍可以参看后面的章节）。所以我们并不能简单地从定义的形式上来判断一个类是新式类还是古典类，而应当通过元类的类型来确定类的类型：古典类的元类为`types.ClassType`，新式类的元类为`type`类。

新式类相对于古典类来说有很多优势：能够基于内建类型构建新的用户类型，支持`property`和描述符特性等。作为新式类的祖先，`Object`类中还定义了一些特殊方法，如：`__new__()`，`__init__()`，`__delattr__()`，`__getattr__()`，`__setattr__()`，`__hash__()`，`__repr__()`，`__str__()`等。`object`的子类可以对这些方法进行覆盖以满足自身的特殊需求，建议62中会对其中某些内容详细阐述，感兴趣的读者可以阅读。



注意

在Python中一切皆对象，`type`也是对象。

[1] 这里的鸿沟指的是：在2.2版本之前，类和类型并不统一，如`a`是古典类`ClassA`的一个实例，那么`a.__class__`返回`'class__main__ClassA'`，`type(a)`返回`<type 'instance'>`。当引入新类后，比如`ClassB`是个新类，`b`是`ClassB`的实例，`b.__class__`和`type(b)`都是返回`'class'__main__.ClassB'`。

建议55: `__init__()`不是构造方法

很多Pythoner会有这样的误解，认为`__init__()`方法是类的构造方法。因为从表面上看它确实很像构造方法：当需要实例化一个对象的时候，使用`a=Class(args...)`便可以返回一个类的实例，其中`args`的参数与`__init__()`方法中声明的参数一样。可是事实真相是怎样的呢？我们通过例子说明。

```
class A(object):
    def __new__(cls, *args, **kwargs):
        print cls
        print args
        print kwargs
        print "-----"
        instance = object.__new__(cls, *args, **kwargs)
        print instance
    def __init__(self, a, b):
        print "init gets called"
        print "self is", self
        self.a, self.b = a, b

a1=A(1,2)
print a1.a
print a1.b
```

运行程序输出如下：

```
<class '__main__.A'>
(1, 2)
{}
-----
<__main__.A object at 0x00D66B30>
Traceback (most recent call last):
File "test.py", line 15, in <module>
print a1.a
AttributeError: 'NoneType' object has no attribute 'a'
```

我们原本期望的是能够正确输出`a`和`b`的值，可是运行却抛出了异常。除了异常外还有来自对`__new__()`方法调用所产生的输出，可是我们明明没有直接调用`__new__()`方法，原因在哪里？实际上`__init__()`并不是真正意义上的构造方法，`__init__()`方法所做的工作是在类的对象

创建好之后进行变量的初始化。`__new__()`方法才会真正创建实例，是类的构造方法。这两个方法都是`object`类中默认的方法，继承自`object`的新式类，如果不覆盖这两个方法将会默认调用`object`中对应的方法。上面的程序抛出异常是因为`__new__()`方法中并没有显式返回对象，因此实际上`a1`为`None`，当去访问实例属性`a`时抛出“`AttributeError: 'NoneType' object has no attribute 'a'`”的错误也就不难理解了。

我们来看看`__new__()`方法和`__init__()`方法的定义。

- `object.__new__(cls[,args...])`：其中`cls`代表类，`args`为参数列表。

- `object.__init__(self[,args...])`：其中`self`代表实例对象，`args`为参数列表。

- 这两个方法之间有些不同点，总结如下：

- 根据Python文档

(http://docs.python.org/2/reference/datamodel.html#object.__new__)可知，`__new__()`方法是静态方法，而`__init__()`为实例方法。

- `__new__()`方法一般需要返回类的对象，当返回类的对象时将会自动调用`__init__()`方法进行初始化，如果没有对象返回，则`__init__()`方法不会被调用。`__init__()`方法不需要显式返回，默认为`None`，否则会在运行时抛出`TypeError`。

- 当需要控制实例创建的时候可使用`__new__()`方法，而控制实例初始化的时候使用`__init__()`方法。

- 一般情况下不需要覆盖`__new__()`方法，但当子类继承自不可变类型，如`str`、`int`、`unicode`或者`tuple`的时候，往往需要覆盖该方法。

·当需要覆盖__new__()和__init__()方法的时候这两个方法的参数必须保持一致，如果不一致将导致异常。示例如下：

```
>>> class Test(object):
...     def __new__(cls,x):
...         return super(Test,cls).__new__(cls)
...     def __init__(self,x,y):
...         self.x=x
...         self.y = y
...
>>> Test(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 3 arguments (2 given)
>>> Test(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() takes exactly 2 arguments (3 given)
>>>
```

前面我们提到，一般情况下覆盖__init__()方法就能满足大部分需求，那么在什么特殊情况下需要覆盖__new__()方法呢？有以下几种情况：

1) 当类继承（如str、int、unicode、tuple或者frozenset等）不可变类型且默认的__new__()方法不能满足需求的时候。来看一个例子：假设我们需要一个不可修改的集合，该集合能够将任何以空格隔开的字符串变为集合中的元素。现在不覆盖__new__()方法，仅覆盖__init__()方法看看是否可行。

```
class UserSet(frozenset):
    def __init__(self, arg=None):
        if isinstance(arg, basestring):
            arg = arg.split()
        frozenset.__init__(self, arg)
print UserSet("I am testing ")
print frozenset("I am testing ")
```

运行程序发现其输出如下：

```
UserSet(['a', ' ', 'e', 'g', 'I', 'm', 'n', 'i', 's', 't'])
frozenset(['a', ' ', 'e', 'g', 'I', 'm', 'n', 'i', 's', 't'])
```

显然没有满足用户的需求，用户希望得到的输出是`UserSet (['T', 'frozen', 'set', 'am', 'tesing'])`。实际上这些不可变类型的`__init__()`方法是个伪方法，必须重新覆盖`__new__()`方法才能满足需求。`__new__()`方法实现的代码如下：

```
def __new__(cls, *args):
    if args and isinstance (args[0], basestring):
        args = (args[0].split (,),) + args[1:]
    return super (UserSet, cls).__new__(cls, *args)
```

2) 用来实现工厂模式或者单例模式或者进行元类编程（元类编程中常常需要使用`__new__()`来控制对象创建。这部分内容会在建议62中进行阐述）的时候。以简单工厂为例子，它由一个工厂类根据传入的参量决定创建出哪一种产品类的实例，属于类的创建型模式。其类的关系如图6-2所示。

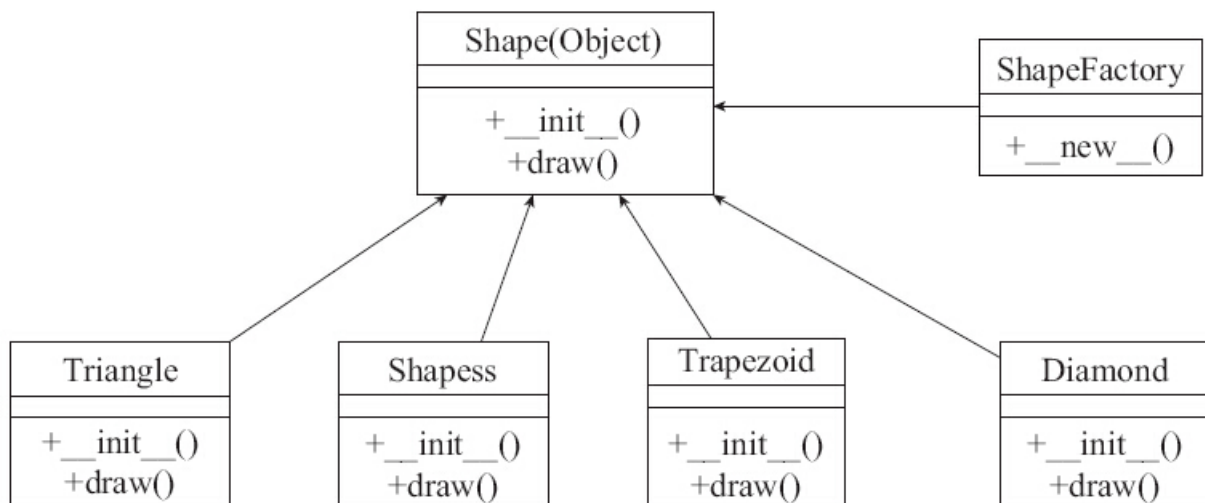


图6-2 简单工厂模式类关系示意图

工厂模式的实现代码如下：

```
class Shape(object):
    def __init__(object):
        pass
    def draw(self):
        pass
```

```

class Triangle(Shape):
    def __init__(self):
        print " I am a triangle"
    def draw(self):
        print "I am drawing triangle"
class Rectangle(Shape):
    def __init__(self):
        print " I am a rectnagle"
    def draw(self):
        print "I am drawing triangle"
class Trapezoid(Shape):
    def __init__(self):
        print " I am a trapezoid"
    def draw(self):
        print "I am drawing triangle"
class Diamond(Shape):
    def __init__(self):
        print " I am a diamond"
    def draw(self):
        print "I am drawing triangle"
class ShapeFactory(object):
    shapes = {'triangle': Triangle, 'rectangle': Rectangle, 'trapezoid':
        Trapezoid, 'diamond': Diamond}
    def __new__(klass, name):
        if name in ShapeFactory.shapes.keys():
            print "creating a new shape %s" % name
            return ShapeFactory.shapes[name]()
        else:
            print "creating a new shape %s" % name
            return Shape()

```

在ShapeFactory类中重新覆盖了__new__()方法，外界通过调用该方法来创建其所需的对象类型，但如果所请求的类是系统所不支持的，则返回Shape对象。在引入了工厂类之后，只需要使用如下形式就可以创建不同的图形对象：

```

ShapeFactory('rectangle').draw()

```

3) 作为用来初始化的__init__()方法在多继承的情况下，子类的__init__()方法如果不显式调用父类的__init__()方法，则父类的__init__()方法不会被调用。

```

    def __init__(self):
        print "I am A's __init__"
class B(A):
    def __init__(self):
        print "I am B's __init__"
b = B()

```

程序输出为：I am B's__init__。父类A的__init__()方法并没有被调用，所以要初始化父类中的变量需要在子类的__init__()方法中使用super (B,self).__init__()。对于多继承的情况，我们可以通过迭代子类的__bases__属性中的内容来逐一调用父类的初始化方法。



注意

__new__()方法才是类的构造方法，而__init__()不是。

建议56：理解名字查找机制

在Python中，所有所谓的变量，其实都是名字，这些名字指向一个或多个Python对象。比如以下代码：

```
>>> a = 1
>>> b = a
>>> c = 'china'
>>> id(a) == id(b)
True
>>> id(a) == id(c)
False
```

从中我们可以看出，名字a和b指向同一个Python对象，即一个int类型的对象，这个对象的值为1；而c则指向另一个Python对象，它是一个str类型的对象。所有的这些名字，都存在于一个表里（又称为命名空间），一般情况下，我们称之为局部变量（locals），可以通过locals()函数调用看到。

```
>>> locals()
{'a': 1, 'c': 'china', 'b': 1, '__builtins__': <module '__builtin__' (built-in)>,
  '__package__': None, '__name__': '__main__', '__doc__': None}
```

现在我们是直接在Python shell中执行这一些代码，实际上这些变量也是全局的，所以在一个叫globals的表里也可以看到。

```
>>> globals()
{'a': 1, 'c': 'china', 'b': 1, '__builtins__': <module '__builtin__' (built-in)>,
  '__package__': None, '__name__': '__main__', '__doc__': None}
```

如果我们在一个函数里面定义这些变量，情况会有所不同。

```
>>> def foo(x):
...     e = 1
...     f = e
```

```

...     g = 'china'
...     print locals()
...     print '#' * 10
...     print globals()
...     print '#' * 10
...     print c
①打印全局变量c
...
>>> foo(1)
{'e': 1, 'g': 'china', 'f': 1, 'x': 1}
#####
{'a': 1, 'c': 'china', 'b': 1, '__builtins__': <module '__builtin__' (built-
in)>, '__package__': None, '__name__': '__main__', 'foo': <function foo at
0x104fad320>, '__doc__': None}
#####
china

```

可以看到函数中的`locals()`返回值并不包含之前定义在全局中的`a`、`b`、`c`等名字，只有定义在函数内的`e`、`f`、`g`和函数形参`x`，这是为什么呢？要回答这个问题，首先要理解Python中变量的作用域。

Python中所有的变量名都是在赋值的时候生成的，而对任何变量名的创建、查找或者改变都会命名空间（`namespace`）中进行。变量名所在的命名空间直接决定了其能访问到的范围，即变量的作用域。Python中的作用域自Python2.2之后分为局部作用域（`local`）、全局作用域（`Global`）、嵌套作用域（`enclosing functions locals`）以及内置作用域（`Build-in`）这4种。

- 局部作用域**：一般来说函数的每次调用都会创建一个新的本地作用域，拥有新的命名空间。因此函数内的变量名可以与函数外的其他变量名相同，由于其命名空间不同，并不会产生冲突。默认情况下函数内部任意的赋值操作（包括`=`语句、`import`语句、`def`语句、参数传递等）所定义的变量名，如果没用`global`语句，则申明都为局部变量，即仅在该函数内可见。

- 全局作用域**：定义在Python模块文件中的变量名拥有全局作用域，需要注意的是这里的全局仅限单个文件，即在一个文件的顶层的变量名仅在这个文件内可见，并非所有的文件，其他文件中想使用这

些变量名必须先导入文件对应的模块。当在函数之外给一个变量名赋值时是在其全局作用域的情况下进行的。

·**嵌套作用域**：一般在多重函数嵌套的情况下才会考虑到。需要注意的是`global`语句仅针对全局变量，在嵌套作用域的情况下，如果想在嵌套的函数内修改外层函数中定义的变量，即使使用`global`进行申明也不能达到目的，其结果最终是在嵌套的函数所在的命名空间中创建了一个新的变量。示例如下：

```
var = 'a'
def inner():
    global var
    var = 'b'
    print 'inside inner, var is ', var
inner()
print 'inside outer function, var is ', var
```

上述程序的输出如下：

```
inside inner, var is  b
inside outer function, var is  a
```

·**内置作用域**：这个相对简单，它是通过一个标准库中名为`__builtin__`的模块来实现的。

回到前面代码中标注①的语句`print c`，仍然正确输出了`china`这个值。这是因为当访问一个变量的时候，其查找顺序遵循变量解析机制LEGB法则，即依次搜索4个作用域：局部作用域、嵌套作用域、全局作用域以及内置作用域，并在第一个找到的地方停止搜寻，如果没有搜到，则会抛出异常。因此当存在多个同名变量的时候，操作生效的往往是搜索顺序在前的。具体来说Python的名字查找机制如下：

1) 在最内层的范围内查找，一般而言，就是函数内部，即在`locals()`里面查找。

2) 在模块内查找，即在`globals()`里面查找。

3) 在外层查找，即在内置模块中查找，也就是在`__builtin__`中查找。

至此，我们可以理解清楚能够在`foo()`函数中访问到名字`c`的原因在于当Python在局部变量中找不到`c`时，它会尝试在模块级的全局变量中查找，并成功地找到该名字。

不过，当我们试图改变全局变量的值时，事情可能跟想象的稍有不同。

```
>>> def bar():
...     c = 'america'
...     print c
...
>>> bar()
america
>>> print c
china
```

真奇怪！不是吗？在`bar()`函数中修改`c`的值，并没有修改到全局变量的`c`，而是好像`bar()`函数有了一个局部变量`c`一样！事实上确实如此，在CPython的实现中，只要出现了赋值语句（或者称为名字绑定），那么这个名字就被当作局部变量来对待。所以在这里如果需要改变全局变量`c`的值，就需要使用`global`关键字。

```
>>> def bar():
...     global c
...     c = 'america'
...     print c
...
>>> bar()
america
>>> print c
america
```

不过，随着更多Python特性的加入，事情变得更加复杂起来。比如在Python闭包中，有这样的问题：

```
>>> def foo():
...     a = 1
...     def bar():
...         b = a * 2
...         a = b + 1
...         print a
...     return bar
...
>>> foo()()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in bar
UnboundLocalError: local variable 'a' referenced before assignment
```

从上例中可以看出，在闭包bar()中，在编译代码为字节码时，因为存在a=b+1这条语句，所以a被当作了局部变量看待，而执行时因为b=a*2先执行，此时局部变量a尚不存在，所以产生了一个UnboundLocalError。在Python2.x中可以使用global关键字解决部分问题，先把a创建为一个模块全局变量，然后在所有读写（包括只是访问）该变量的作用域中都要先使用global声明其为全局变量。

```
>>> a = 1
>>> def foo(x):
...     global a
...     a = a * x
...     def bar():
...         global a
...         b = a * 2
...         a = b + 1
...         print a
...     return bar
...
>>> foo(1)()
3
```

这种方案抛开编程语言并不提倡全局变量不谈，有的时候还影响业务逻辑。此外，还有把a作为容器的一个元素来对待的方案，但也都相当复杂。真正的解决方案是Python3引入的nonlocal关键字，通过它能够解决这方面的问题。

```
>>> def foo(x):
...     a = x
...     def bar():
...         nonlocal a
...         b = a * 2
...         a = b + 1
...         print(a)
...     return bar
...
>>> bar1 = foo(1)
```

建议57：为什么需要self参数

self想必大家都不陌生，在类中当定义实例方法的时候需要将第一个参数显式声明为**self**，而调用的时候并不需要传入该参数。我们可以使用**self.x**来访问实例变量，也可以在类中使用**self.m()**来访问实例方法。**self**的使用示例如下：

```
class SelfTest(object):
    def __init__(self, name):
        self.name = name
    def showself(self):
        print "self here is%s"%self
    def display(self):
        self.showself()
        print ("The name is:", self.name)
st = SelfTest("instance self")
st.display()
print "%X"%(id(st))
```

上例中我们使用**self.name**来表示实例变量**name**，在**display**方法中使用**self.showself()**来调用实例方法**showself()**，并且调用的时候没有显式传入**self**参数。程序输出如下：

```
self here is<__main__.SelfTest object at 0x00D67C10>
('The name is:', 'instance self')
D67C10
```

从上述输出中可以看出，**self**表示的就是实例对象本身，即**SelfTest**类的对象在内存中的地址。**self**是对对象**st**本身的引用。我们在调用实例方法的时候也可以直接传入实例对象：如：

SelfTest.display(st)。其实**self**本身并不是Python的关键字（**cls**也不是），可以将**self**替换成任何你喜欢的名称，如**this**、**obj**等，实际效果和**self**是一样的（并不推荐这样做，使用**self**更符合约定俗成的原则）。

也许很多人感受self最奇怪的地方就是：在方法声明的时候需要定义self作为第一个参数，而调用方法的时候却不用传入这个参数。虽然这并不影响语言本身的使用，而且也很容易遵循这个规则，但多多少少会在心里问一问：既然这样，为什么必须在定义方法的时候声明self参数呢？去掉第一个参数self不是更简洁吗？就如C++中的this指针一样。我们来简单探讨一下为什么需要self。

1) Python在当初设计的时候借鉴了其他语言的一些特征，如Modula-3中方法会显式地在参数列表中传入self。Python起源于20世纪80年代末，那个时候的很多语言都有self，如Smalltalk、Modula-3等。Python在最开始设计的时候受到了其他语言的影响，因此借鉴了其中的一些理念（注：即使不了解Smalltalk、Modula-3也没有关系，此处只是为了说明当初在设计Python时借鉴了其他语言的一些特点）。下面这段话摘自Guido 1998年接受的一个访问，他自己也提到了这一点。

Andrew :What other languages or systems have influenced Python's design?（在Python的设计过程中受到了哪些语言或者系统的影响？）

Guido :There have been many.ABC was a major influence,of course,since I had been working on it at CWI.It inspired the use of indentation to delimit blocks,which are the high-level types and parts of object implementation.I'd spent a summer at DEC's Systems Research Center,where I was introduced to Modula-2+;theModula-3final report was being written there at about the same time. What I learned there showed up in Python's exception handling,modules,and the fact that methods explicitly contain “self” in their parameter list. String slicing came from Algol-68 and Icon.（有很多，当然，ABC影响最大，因为在CWI的时候我一直在研究它。它启发了我使用缩进来分块，这些是高级的类型以及部分对象

的实现。我在DEC的系统研究中心花费了一个暑假的时间，在那里我学到了Modula-2+。而Modula-3也是在同一时期在那里被实现的。我在那里学到的，在Python的异常处理、模块以及方法的参数列表中显式包含self中都有体现，而字符串的分隔则是从Algol-68和Icon中借鉴的。)

2) Python语言本身的动态性决定了使用self能够带来一定便利。下例中len表示求点到原点距离的函数，现在有表示直角三角形的类Rtriangle，我们发现求第三边边长和len所实现的功能其实是一样的，所以打算直接重用该方法。由于self在函数调用中是隐式传递的，因此当直接调用全局函数len()时传入的是point对象，而当在类中调用该方法时，传入的是类所对应的对象，使用self可以在调用的时候决定应该传入哪一个对象。

```
>>> def len(point):
...     return math.sqrt(point.x ** 2 + point.y ** 2)
...
>>> class RTriangle(object):
...     def __init__(self, right_angle_sideX, right_angle_sideY):
...         self.right_angle_sideX = right_angle_sideX
...         self.right_angle_sideY = right_angle_sideY
...
>>> RTriangle.len = len
>>> rt = RTriangle(3,4)
>>> rt.len()
5.0
>>>
```

Python属于一级对象语言（first class object），如果m是类A的一个方法，有好几种方式都可以引用该方法，如下例所示：

```
>>> class A:
...     def m(self,value):
...         pass
...
>>> A.__dict__['m']
<function m at 0x00D617B0>
>>> A.m.__func__
<function m at 0x00D617B0>
```

实例方法是作用于对象的，如果用户使用上述两种形式来调用方法，最简单的方式就是将对象本身传递到该方法中去，**self**的存在保证了A.__dict__['m'](a,2)的使用和a.(2)一致。同时当子类覆盖了父类中的方法但仍然想调用该父类的方法的时候，可以方便地使用 **baseclass.methodname (self, <argument list>)**或**super (childclass, self) .methodname (< argument list >)**来实现。

3) 在存在同名的局部变量以及实例变量的情况下使用**self**使得实例变量更容易被区分。

```
value = 'default global'
class Test(object):
    def __init__(self, par1):
        value = par1+"-----"
        self.value = value
    def show(self):
        print self.value
        print value
a = Test("instance")
show()
```

上述程序中第一个**value**为全局变量，第二个为局部变量，仅仅在方法中可见，而类同时定义了一个实例变量**value**。如果没有**self**，我们很难区分到底哪个表示局部变量，哪个表示实例变量，有了**self**这一切一目了然了。

其实关于要不要**self**或者要不要将**self**作为关键字一直是一个很有争议的问题，也有人提了一些修正建议。但Guido认为，基于Python目前的一些特性（如类中动态添加方法，在类风格的装饰器中没有**self**无法确认是返回一个静态方法还是类方法等）保留其原有设计是个更好的选择，更何况Python的哲学是：显式优于隐式（Explicit is better than implicit.）。个人体会是除了在定义的时候多输入几个字符外，**self**的存在并不会给用户带来太多困扰，我们没有必要过分纠结于它是否存在这个问题。

建议58：理解MRO与多继承

跟其他编程语言一样，Python也支持多继承。多继承的语法非常简单。

```
class DerivedClassName(Base1, Base2, Base3)
```

谈到多继承，我们来讨论一下图6-3所示的菱形继承的经典问题。

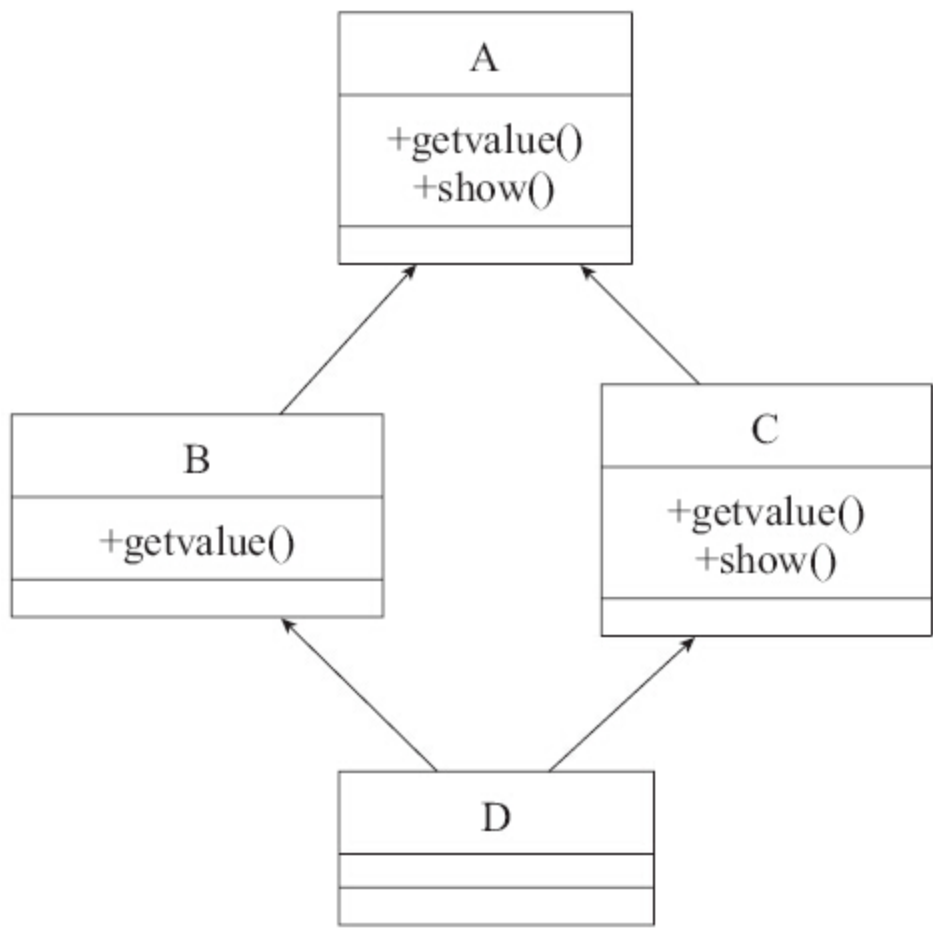


图6-3 多继承UML示意图

假设有图6-3所示继承关系，当用古典类实现的时候，如果有实例d=D()，当调用d.getvalue()和d.show()方法的时候分别对应哪个父类中的方法？当改为新式类来实现时，结果又将是怎样的呢？我们来看具体实现：

```
class A():
    def getvalue(self):
        print "return value of A"
    def show(self):
        print "I can show the information of A"
class B(A):
    def getvalue(self):
        print "return value of B"
class C(A):
    def getvalue(self):
        print "return value of C"
    def show(self):
        print "I can show the information of C"
class D(B,C):pass
```

当用古典类实现的时候我们会发现，分别调用的是B类的getvalue()方法和A类中的show()方法，而当改为新式类实现（请读者自行验证）的时候，结果却变为调用B类的getvalue()方法和C类的show()方法。从两种不同实现方式的输出上也可以证实这一点。

古典类输出如下：

```
return value of B
I can show the information of A
```

新式类输出如下：

```
return value of B
I can show the information of C
```

为什么两种情况下输出结果会有所不同呢？这背后到底发生了什么？根本原因在哪里？实际上，导致这些不同点的根本原因在于古典

类和新式类之间所采取的MRO（Method Resolution Order，方法解析顺序）的实现方式存在差异。

在古典类中，MRO搜索采用简单的自左至右的深度优先方法，即按照多继承申明的顺序形成继承树结构，自顶向下采用深度优先的搜索顺序，当找到所需要的属性或者方法的时候就停止搜索。因此如图6-3所示，当调用d.getvalue()的时候，其搜索顺序为D->B，所以调用的是B类中对应的方法。而d.show()的搜索顺序为D->B->A，因此最后调用的是A类中对应的方法。

而新式类采用的是C3 MRO搜索方法，该算法描述如下：

假定，C1C2...CN表示类C1到CN的序列，其中序列头部元素(head)=C1，序列尾部(tail)定义为=C2..CN；

C继承的基类自左向右分别表示为B1，B2...BN；

L[C]表示C的线性继承关系，其中L[object]=object。

算法具体过程如下：

$$L[C(B1 \dots BN)] = C + \text{merge}(L[B1] \dots L[BN], B1 \dots BN)$$

其中merge方法的计算规则如下：在L[B1]...L[BN],B1...BN中，取L[B1]的head，如果该元素不在L[B2]...L[BN],B1...BN的尾部序列中，则添加该元素到C的线性继承序列中，同时将该元素从所有列表中删除（该头元素也叫good head），否则取L[B2]的head。继续相同的判断，直到整个列表为空或者没有办法找到任何符合要求的头元素（此时将引发一个异常）。

我们结合上面的例子来说明C3 MRO算法的具体计算方法，以新式类实现的上述菱形继承关系如图6-4所示。

根据算法规则有如下关系表达式：

```
L(O)=O
; L(A)=AO
;
```

则：

```
L(B)=B+merge(L(A))=B+merge(AO)=B+A+merge(O,O)=B,A,O
L(C)=C+merge(L(A))=C+merge(AO)=C+A+merge(O,O)=C,A,O
L(D)=D+merge(L(B),L(C),BC)
    =D+merge(BAO,CAO,BC)
    =D+B+merge(AO,CAO,C)(
下一个计算取AO
的头A
, 但A
包含在CAO
的尾部, 因此不满足条件,
跳到下一个元素CAO
继续计算)
    =D+B+C+merge(AO,AO)
    =D+B+C+A+O =DBCAO
```

因此对于上述例子，当D的实例d调用getvalue()和show()方法时按照D->B->C->A->O的顺序进行搜索，在第一个找到该方法的位置停止搜索并调用该类对应的方法，因此getvalue()会在B的类中找到对应的方法，而show会在C()类中找到对应的方法。

关于MRO的搜索顺序我们也可以在新式类中通过查看__mro__属性得到证实。D.__mro__的输出如下：

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__mai
n__.A'>, <type 'object'>)
```

实际上MRO虽然叫方法解析顺序，但它不仅是针对方法搜索，对于类中的数据属性也适用。读者可以自行验证。

根据C3 MRO算法的描述，如果找不到满足条件的good head，则会摒弃该元素从而对下一个元素进行查找。但如果找遍了所有的元素都找不到符合条件的good head会怎么样呢？来看一个具体例子。

```
class A(object): pass
class B(object): pass
class C(A, B): pass    ... ..
①基类顺序为A
, B
class D(B, A): pass    ... ..
②基类顺序为B
, A
class E(C, D): pass
```

运行程序我们会发现这种情况下有异常抛出。

```
TypeError: Error when calling the metaclass bases
  Cannot create a consistent method resolution
order (MRO) for bases B, A
```

根据上述代码的继承关系图（请读者自行画出）和MRO算法可以得出：

```
L(E)=E+merge(L(C), L(D), CD)
    =E+merge(CABO, CBAO, CD)
    =E+C+merge(ABO, BAO, D)
    =E+C+D+merge(ABO+BAO)
```

当算法进行到最后一步的时候便再也找不到满足条件的head了，因为当选择ABO的头A元素的时候，发现其包含在BAO的尾部AO中；同理，B包含在BO中，此时便形成了一个死锁，Python解释器此时不知道如何处理这种情况，便直接抛出异常，这就是上述例子有异常抛出的原因。

菱形继承是我们在多继承设计的时候需要尽量避免的一个问题。

建议59：理解描述符机制

除了在不同的局部变量、全局变量中查找名字，还有一个相似的场景不可不察，那就是查找对象的属性。在Python中，一切皆是对象，所以类也是对象，类的实例也是对象。

```
>>> class MyClass(object):
...     class_attr = 1
...
>>> MyClass.__dict__
dict_proxy({'__dict__': <attribute '__dict__' of 'MyClass' objects>, '
__module__': '__main__', '__weakref__': <attribute '__weakref__'
of 'MyClass' objects>, '__doc__': None, 'class_attr': 1})
```

每一个类都有一个__dict__属性，其中包含的是它的所有属性，又称为类属性。留意类属性的最后一个元素，可以看到我们代码中定义的属性在其中的体现。

```
>>> my_instance = MyClass()
>>> my_instance.__dict__
{}
```

除了与类相关的类属性之外，每一个实例也有相应的属性表（__dict__），称为实例属性。当我们通过实例访问一个属性时，它首先会尝试在实例属性中查找，如果找不到，则会到类属性中查找。

```
>>> my_instance.class_attr
1
```

可以看到实例my_instance可以访问类属性class_attr。但与读操作有所不同，如果通过实例增加一个属性，只能改变此实例的属性，对类属性而言，并没有丝毫变化。这从下面的代码中可以得到印证。

```
>>> my_instance.inst_attr = 'china'
>>> my_instance.__dict__
{'inst_attr': 'china'}
>>> MyClass.__dict__
dict_proxy({'__dict__': <attribute '__dict__' of 'MyClass' objects>, '
__module__': '__main__', '__weakref__': <attribute '__weakref__'
of 'MyClass' objects>, '__doc__': None, 'class_attr': 1})
```

那么，能不能给类增加一个属性呢？答案是，能，也不能。说能，是因为每一个class也是一个对象，动态地增减对象的属性与方法正是Python这种动态语言的特性，自然是支持的。

```
>>> MyClass.class_attr2 = 100
>>> my_instance.class_attr2
100
```

说不能，是因为在Python中，内置类型和用户定义的类型是有分别的，内置类型并不能够随意地为它增加属性或方法。

```
>>> str.new_attr = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't set attributes of built-in/extension type 'str'
>>> setattr(str, 'new_attr', 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't set attributes of built-in/extension type 'str'
```

至此，我们应当理解了，当我们通过“.”操作符访问一个属性时，如果访问的是实例属性，与直接通过__dict__属性获取相应的元素是一样的；而如果访问的是类属性，则并不相同：“.”操作符封装了对两种不同属性进行查找的细节。

```
>>> my_instance.__dict__['inst_attr']
'china'
>>> my_instance.__dict__['class_attr2']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'class_attr2'
```

不过，这里要讲的并不止于此，“.”操作符封装了对实例属性和类属性查找的细节，只讲了一半事实，还有一部分隐而未谈，那就是描述符机制。

```
>>> MyClass.__dict__['inst_attr']
'china'
>>> MyClass.inst_attr
'china'
```

我们已经知道访问类属性时，通过__dict__访问和使用“.”操作符访问是一样的，但如果是方法，却又不是如此了。

```
>>> class MyClass(object):
...     def my_method(self):
...         print 'my_method'
...
>>> MyClass.__dict__['my_method']
<function my_method at 0x102773aa0>
>>> MyClass.my_method
<unbound method MyClass.my_method>
```

甚至它们的类型都不一样！

```
>>> type(MyClass.my_method)
<type 'instancemethod'>
>>> type(MyClass.__dict__['my_method'])
<type 'function'>
```

这其中作怪的就是描述符了。当通过“.”操作符访问时，Python的名字查找并不是之前说的先在实例属性中查找，然后再在类属性中查找那么简单，实际上，根据通过实例访问属性和根据类访问属性的不同，有以下两种情况：

一种是通过实例访问，比如代码obj.x，如果x是一个描述符，那么__getattr__()会返回type(obj).__dict__[x].__get__(obj,type(obj))结果，即：type(obj)获取obj的类型；type(obj).__dict__[x]返回的是一个

描述符，这里有一个试探和判断的过程；最后调用这个描述符的`__get__()`方法。

另一种是通过类访问的情况，比如代码`cls.x`，则会被`__getattrute__()`转换为`cls.__dict__['x'].__get__(None,cls)`。

至此，就能够明白`MyClass.__dict__['my_method']`返回的是`function`而不是`instancemethod`了，原因是没有调用它的`__get__()`方法。是否如此呢？怎么验证一下？我们可以尝试手动调用`__get__()`。

```
>>> t = f.__get__(None, MyClass)
>>> t
<unbound method MyClass.my_method>
>>> type(t)
<type 'instancemethod'>
```

看，果然是这样！这是因为描述符协议是一个**Duck Typing**的协议，而每一个函数都有`__get__`方法，也就是说其他每一个函数都是描述符。

描述符机制有什么作用呢？其实它的作用编写一般程序的话还真用不上，但对于编写程序库的读者来说，就非常有用了。比如大家熟悉的已绑定方法和未绑定方法，它是怎么来的呢？

```
>>> MyClass.my_method
<unbound method MyClass.my_method>
>>> a = MyClass()
>>> a.my_method
<bound method MyClass.my_method of <__main__.MyClass object at 0x10277a490>>
```

上面例子输出的不同，其实来自于对描述符的`__get__()`的调用参数的不同，当以`obj.x`的形式访问时，调用参数是`__get__(obj,type(obj))`；而以`cls.x`的形式访问时，调用参数是

`__get__(None, type(obj))`，这可以通过未绑定方法的`im_self`属性为`None`得到印证。

```
>>> print MyClass.my_method.im_self
None
>>> a.my_method.im_self
<__main__.MyClass object at 0x10277a490>
```

除此之外，所有对属性、方法进行修饰的方案往往都用到了描述符，比如`classmethod`、`staticmethod`和`property`等。在这里，给出`property`的参考实现作为本节的结束，更深入的应用可以进一步参考Python源码中的其他用法。

```
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc
    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError, "unreadable attribute"
        return self.fget(obj)
    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError, "can't set attribute"
        self.fset(obj, value)
    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError, "can't delete attribute"
        self.fdel(obj)
```

建议60: 区别__getattr__()和__getattribute__()方法

__getattr__()和__getattribute__()都可以用做实例属性的获取和拦截（注意，仅对实例属性（instance variable）有效，非类属性），__getattr__()适用于未定义的属性，即该属性在实例中以及对应的类的基类以及祖先类中都不存在，而__getattribute__()对于所有属性的访问都会调用该方法。它们的函数签名分别为：

```
__getattr__: __getattr__(self,name)
__getattribute__: __getattribute__(self,name)
```

其中参数name为属性的名称。需要注意的是__getattribute__()仅应用于新式类。

既然这两种方法都用作属性的访问，那么它们有什么区别呢？我们来看一个例子。

```
class A(object):
    def __init__(self,name):
        self.name = name
a = A("attribute")
print a.name
print a.test
```

上面的程序输出如下：

```
attribute
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print a.test
AttributeError: 'A' object has no attribute 'test'
```

当访问一个不存在的实例属性的时候就会抛出AttributeError异常。这个异常是由内部方法__getattribute__(self,name)抛出的，因为__getattribute__()会被无条件调用，也就是说只要涉及实例属性的访问就会调用该方法，它要么返回实际的值，要么抛出异常。Python的文档<http://docs.python.org/2/reference/datamodel.html#object.getattribute>中也提到了这一点。那么__getattr__()会在什么情况下调用呢？我们在上面的例子中添加__getattr__()方法试试。

```
def __getattr__(self,name):  
    print ("calling __getattr__:",name)
```

再次运行程序会发现输出为：

```
attribute  
( 'calling __getattr__:', 'test')  
None
```

这次程序没有抛出异常，而是调用了__getattr__()方法。实际上__getattr__()方法仅如下情况下才被调用：属性不在实例的__dict__中；属性不在其基类以及祖先类的__dict__中；触发AttributeError异常时（注意，不仅仅是__getattribute__()引发的AttributeError异常，property中定义的get()方法抛出异常的时候也会调用该方法）。需要特别注意的是当这两个方法同时被定义的时候，要么在__getattribute__()中显式调用，要么触发AttributeError异常，否则__getattr__()永远不会被调用。__getattribute__()及__getattr__()方法都是Object类中定义的默认方法，当用户需要覆盖这些方法时有以下几点注意事项：

- 1) 避免无穷递归。当在上述例子中添加__getattribute__()方法后程序运行会抛出RuntimeError异常提示“RuntimeError:maximum recursion depth exceeded.”。

```
def __getattr__(self, attr):
    try:
        return self.__dict__[attr]
    except KeyError:
        return 'default'
```

这是因为属性的访问调用的是覆盖了的`__getattr__()`方法，而该方法中`self.__dict__[attr]`又要调用`__getattr__(self,attr)`，于是产生了无穷递归，即使将语句`self.__dict__[attr]`替换为`self.__getattr__(self,attr)`和`getattr(self,attr)`也不能解决问题。正确的做法是使用`super(obj,self).__getattr__(attr)`，因此上面的例子可以改为：`super(A,self).__getattr__(attr)`或者`object.__getattr__(self,attr)`。无穷递归是覆盖`__getatt__()`和`__getattr__()`方法的时候需要特别小心。

2) 访问未定义的属性。如果在`__getattr__()`方法中不抛出`AttributeError`异常或者显式返回一个值，则会返回`None`，此时可能会影响到程序的运行预期。我们来看一个示例：

```
class A(object):
    def __init__(self, name):
        self.name = name
        self.x = 20
    def __getattr__(self, name):
        print ("calling __getattr__:", name)
        if name == 'z':
            return self.x ** 2
        elif name == 'y':
            return self.x ** 3
    def __getattr__(self, attr):
        try:
            return super(A, self).__getattr__(attr)
        except KeyError:
            return 'default'

a = A("attribute")
print a.name
print a.z
if hasattr(a, 't'):
    c = a.t
    print c
else:
    print "instance a has no attribute t"
```

用户本来的意图是：如果t不属于实例属性，则打印出警告信息，否则给c赋值。按照用户的理解本来应该是输出警告信息的，可是实际却输出None。这是因为在__getattr__()方法中没有抛出任何异常也没有显式返回一个值，None被作为默认值返回并动态添加了属性t，因此hasattr(object,name)的返回结果是True。如果我们在上述例子中抛出异常（raise TypeError('unknown attr:' + name)），则一切将如用户期待的那样。

另外关于__getattr__()和__getattribute__()有以下两点提醒：

1) 覆盖了__getattribute__()方法之后，任何属性的访问都会调用用户定义的__getattribute__()方法，性能上会有所损耗，比使用默认的方法要慢。

2) 覆盖的__getattr__()方法如果能够动态处理事先未定义的属性，可以更好地实现数据隐藏。因为dir()通常只显示正常的属性和方法，因此不会将该属性列为可用属性，上述例子中如果动态添加属性y，即使hasattr(a,'y')的值为True，dir(a)得到的却是如下输出：

```
['_class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattr__',
 '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', 'name', 'x']
```

再来思考一个问题：我们知道property也能控制属性的访问，如果一个类中同时定义了property、__getattribute__()以及__getattr__()来对属性进行访问控制，那么具体的查找顺序是怎样的呢？

```
class A(object):
    _c = "test"
    def __init__(self):
        self.x = None
    @property
    def a(self):
        print "using property to access attribute"
        if self.x is None:
```

```

        print "return value"
        return 'a'
    else:
        print "error occured"
        raise AttributeError
@a.setter
def a(self,value):
    self.x = value
def __getattr__(self, name):
    print "using __getattr__ to access attribute"
    print('attribute name: ', name)
    return "b"
def __getattribute__(self, name):
    print "using __getattribute__ to access attribute"
    return object.__getattribute__(self,name)
a1 = A()
print a1.a
print "-----"
a1.a = 1
print a1.a
print "-----"
print A._c

```

上述程序的输出如下:

```

using __getattribute__ to access attribute
using property to access attribute
using __getattribute__ to access attribute
return value
a
-----
using __getattribute__ to access attribute
using property to access attribute
using __getattribute__ to access attribute
error occured
using __getattr__ to access attribute
('attribute name: ', 'a')
b
-----
test

```

当实例化a1时由于其默认的属性x为None，当我们访问a1.a时，最先搜索的是__getattribute__()方法，由于a是一个property对象，并不存在于a1的dict中，因此并不能返回该方法，此时会搜索property中定义的get()方法，所以返回的结果是a。当用property中的set()方法对x进行修改并再次访问property的get()方法时会抛出异常，这种情况下会触发对__getattr__()方法的调用并返回结果b。程序最后访问类变量输出test是为了说明对类变量的访问不会涉及__getattribute__()和__getattr__()方法。



注意

`__getattribute__()`总会被调用，而`__getattr__()`只有在`__getattribute__()`中引发异常的情况下才会被调用。

建议61：使用更为安全的property

`property`是用来实现属性可管理性的**built-in**数据类型（注意：很多地方将`property`称为函数，我个人认为这是不恰当的，它实际上是一种实现了`__get__()`、`__set__()`方法的类，用户也可以根据自己的需要定义个性化的`property`），其实质是一种特殊的数据描述符（数据描述符：如果一个对象同时定义了`__get__()`和`__set__()`方法，则称为数据描述符，如果仅定义了`__get__()`方法，则称为非数据描述符）。它和普通描述符的区别在于：普通描述符提供的是一种较为低级的控制属性访问的机制，而`property`是它的高级应用，它以标注库的形式提供描述符的实现，其签名形式为：

```
property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
```

`Property`常见的使用形式有以下几种。

1) 第一种形式如下：

```
class Some_Class(object):
    def __init__(self):
        self._somevalue = 0
    def get_value(self):
        print "calling get method to return value"
        return self._somevalue
    def set_value(self, value):
        print "calling set method to set value"
        self._somevalue = value
    def del_attr(self):
        print "calling delete method to delete value"
        del self._somevalue
    x = property(get_value, set_value, del_attr, "I'm the 'x' property.")
obj = Some_Class()
obj.x = 10
print obj.x + 2
del obj.x
obj.x
```

2) 第二种形式如下:

```
class Some_Class(object):
    __x=None
    def __init__(self):
        self.__x = None
    @property
    def x(self):
        print "calling get method to return value"
        return self.__x
    @x.setter
    def x(self,value):
        print "calling set method to set value"
        self.__x = value
    @x.deleter
    def x(self):
        print "calling delete method to delete value"
        del self.__x
```

在了解完property的基本知识之后来探讨一下这些问题: property到底有什么优势呢? 为什么要有这个特性呢? property的优势可以简单地概括为以下几点:

1) 代码更简洁, 可读性更强。这条优势是显而易见的, 显然obj.x+=1比obj.set_value(obj.get_value()+1)要更简洁易读, 而且对于编程人员来说还少敲了几次键盘。

2) 更好的管理属性的访问。property将对属性的访问直接转换为对对应的get、set等相关函数的调用, 属性能够更好地被控制和管理, 常见的应用场景如设置校验(如检查电子邮件地址是否合法)、检测赋值的范围(如某个变量的赋值范围必须在0到10之间)以及对某个属性进行二次计算之后再返回给用户(如将RGB形式表示的颜色转换为#*****形式返回给用户)或者计算某个依赖于其他属性的属性。来看一个使用property控制属性访问的例子。

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
class Date(object):
    def __init__(self,year,month,day):
        self.year = year
        self.month = month
        self.day = day
```

```

def get_date(self):
    return self.year+"-"+self.month+"-"+self.day
def set_date(self,date_as_string):
    year, month, day = date_as_string.split('-')
    if not (2000 <= year <=2015 and 0 <= month <= 12 and 0
        <= day <= 31):
        print "year should be in [2000:2015]"
        print "month should be in [0:12]"
        print "day should be in [0,31]"
        raise AssertionError
    self.year = year
    self.month = month
    self.day = day
date =property(get_date,set_date)

```

创建一个property实际上就是将其属性的访问与特定的函数关联起来，相对于标准属性的访问，其工作原理如图6-5所示。property的作用相当于一个分发器，对某个属性的访问并不直接操作具体的对象，而对标准属性的访问没有中间这一层，直接访问存储属性的对象。

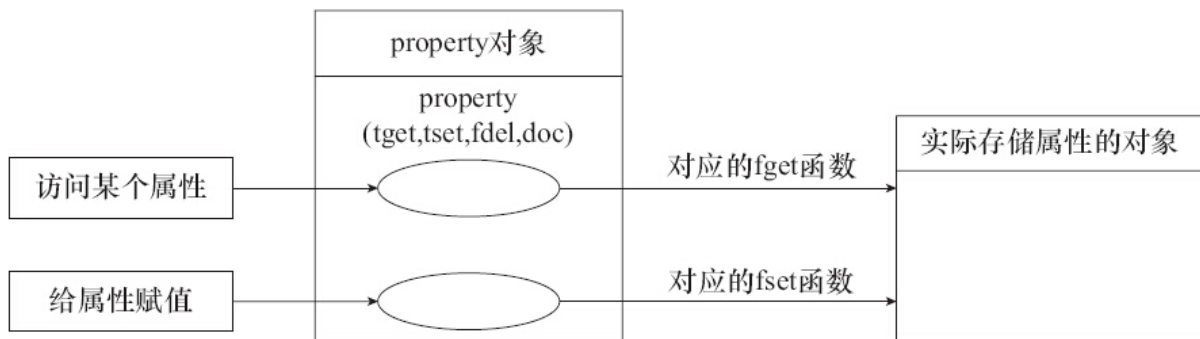


图6-5 property的工作原理

3) 代码可维护性更好。property对属性进行再包装，以类似于接口的形式呈现给用户，以统一的语法来访问属性，当具体实现需要改变的时候（如改变某个内部变量，或者赋值或取值的计算发生改变），访问的方式仍然可以保留一致。例如上述例子中，如果要更改date的显示方式，如“2012年4月20日”，则只需要对get_value()做对应的修改即可，外部程序访问date的方式并不需要改变，因此代码的可维护性大大提高。

4) 控制属性访问权限，提高数据安全性。如果用户想设置某个属性为只读，我们来看看使用property如何满足这个需求。

```
class PropertyTest(object):
    def __init__(self):
        self.__var1=20
    @property
    def x(self):
        return self.__var1
pt = PropertyTest()
print pt.x
pt.x=12
```

上面的程序输出如下：

```
20
Traceback (most recent call last):
File "tests.py", line 11, in <module>
pt.x=12
AttributeError: can't set attribute
```

在前面的代码中我们只实现了get()方法，没有实现set()方法。如果使用第一种形式的property，也只需要设置x=property(get_value)后实现对应的get()方法即可。

值得注意的是：使用property并不能真正完全达到属性只读的目的，正如以双下划线命令的变量并不是真正的私有变量一样，这些方法只是在直接修改属性这条道路上增加了一些障碍。如果用户想访问私有属性，同样能够实现，如上例便可以使用pt._PropertyTest__var1=30来修改属性。那么究竟怎样才能实现真正意义上的只读和私有变量呢？本节最后会探讨这个问题，这里请读者先思考一下。

我们在本节开头提到property本质并不是函数，而是特殊类，既然是类的话，那么就可以被继承，因此用户便可以根据自己的需要定义property。来看以下具体实现：

```

def update_meta (self, other):
    self.__name__ = other.__name__
    self.__doc__ = other.__doc__
    self.__dict__.update(other.__dict__)
    return self
class UserProperty (property):
    def __new__(cls, fget=None, fset=None, fdel=None, doc=None):
        if fget is not None:
            def __get__(obj, objtype=None, name=fget.__name__):
                fget = getattr(obj, name)
                print "fget name:"+fget.__name__
                return fget()
            fget = update_meta(__get__, fget)
        if fset is not None:
            def __set__(obj, value, name=fset.__name__):
                fset = getattr(obj, name)
                print "fset name:"+fset.__name__
                print "setting value:" +str(value)
                return fset(value)
            fset = update_meta(__set__, fset)
        if fdel is not None:
            def __delete__(obj, name=fdel.__name__):
                fdel = getattr(obj, name)
                print "warning: you are deleting attribute
                    using fdel.__name__"
                return fdel()
            fdel = update_meta(__delete__, fdel)
        return property(fget, fset, fdel, doc)
class C(object):
    def get(self):
        print 'calling C.getx to get value'
        return self._x
    def set(self, x):
        print 'calling C.setx to set value'
        self._x = x
    def delete(self):
        print 'calling C.delx to delete value'
        del self._x
    x = UserProperty(get, set, delete)
c = C()
c.x = 1
print c.x
del c.x

```

上述例子中UserProperty继承自property，其构造函数__new__(cls, fget=None, fset=None, fdel=None, doc=None)中重新定义了fget()、fset()以及fdel()方法以满足用户特定的需要，最后返回的对象实际还是property的实例，因此用户能够像使用property一样使用UserProperty。

回到前面的问题：使用property并不能真正完全达到属性只读的目的，用户仍然可以绕过阻碍来修改变量。那么要真正实现只读属性怎么做呢？我们来看一个可行的实现：

```
def ro_property(obj, name, value):
    setattr(obj.__class__, name, property(lambda obj: obj.__dict__["_" + name]))
    setattr(obj, "_" + name, value)
class ROClass(object):
    def __init__(self, name, available):
        ro_property(self, "name", name)
        self.available = available
a = ROClass("read only", True)
print a.name
a._Article__name = "modify"
print a.__dict__
print ROClass.__dict__
print a.name
```

上述程序的输出如下:

```
read only
{'available': True, '__name': 'read only', '_Article__name': 'modify'}
{'__module__': '__main__', 'name': <property object at 0x00CA7330>, '__dict__':
<attribute '__dict__' of 'ROClass' objects>, '__weakref__': <attribute '
__weakre
f__' of 'ROClass' objects>, '__doc__': None, '__init__': <function __init__ at 0
x00D617B0>}}
read only
```

我们发现，当用户再试图用`a._Article__name`来修改变量`_name`的时候并没有达到目的，而是重新创建了新的属性`_Article__name`，这样就能够很好地保护可读属性不被修改，以免造成损失了。

建议62：掌握metaclass

什么是元类（**metaclass**）？也许我们对下面这些说法都不陌生：

- 元类是关于类的类，是类的模板。
- 元类是用来控制如何创建类的，正如类是创建对象的模板一样。
- 元类的实例为类，正如类的实例为对象。

这些说法都没有错，在概念之外我们来进行一些更深入的探讨：元类是如何来控制类的创建的？用户该如何定义自己的元类？在哪些情况下需要用到元类？使用元类可以解决什么问题？

我们知Python中一切皆对象，类也是对象，可以在运行的时候动态创建。当使用关键字**class**的时候，Python解释器在执行的时候就会创建一个对象（这里的对象是指类而非类的实例）。

```
>>> def dynamic_class(name):
...     if name == 'A':
...         class A(object):
...             pass
...         return A
...     else:
...         class B(object):
...             pass
...         return B
...
>>>
>>> UserClass = dynamic_class('A')
>>> print UserClass
<class '__main__.A'>
>>> UserClass()
<__main__.A object at 0x00D67CF0>
>>>
```

既然类是对象，那么它就有其所属的类型，也一定还有什么东西能够控制它的生成。通过**type**查看会发现UserClass的类型为**type**，而其

对象UserClass()的类型为类A。

```
>>> type(UserClass)
<type 'type'>
>>> type(UserClass())
<class '__main__.A'>
```

同时我们知道type还可以这样使用：

```
type(
    类名,
    父类的元组（针对继承的情况，可以为空），
    包含属性的字典（名称和值））
```

例如：

```
>>> A=type('A',(object,),{'value':2})
>>> A.value
>>> print A
<class '__main__.A'>
>>> class C(A):
...     pass
...
>>> print C
<class '__main__.C'>
>>>
>>> print C.__class__
<type 'type'>
```

上例中type通过接受类的描述作为参数返回一个对象，这个对象可以被继承，属性能够被访问，它实际是一个类，其创建由type控制，由type所创建的对象__class__属性为type。type实际上是Python的一个内建元类，用来直接指导类的生成。当然，除了使用内建元类type，用户也可以通过继承type来自定义元类。我们来看一个利用元类实现强制类型检查的例子。

```
class TypeSetter(object):
    def __init__(self,fieldtype):
        print "set attribute type",fieldtype
        self.fieldtype = fieldtype
    def is_valid(self,value):
        return isinstance(value,self.fieldtype)
```

```

class TypeCheckMeta(type):
    def __new__(cls, name, bases, dict):
        print '-----'
        print "Allocating memory for class", name
        print name
        print bases
        print dict
        return super(TypeCheckMeta, cls).__new__(cls, name, bases, dict)
    def __init__(cls, name, bases, dict):
        cls._fields = {}
        for key, value in dict.items():
            if isinstance(value, TypeSetter):
                cls._fields[key] = value
def sayHi(cls):
    print "Hi"

```

TypeSetter用来设置属性的类型，TypeCheckMeta为用户自定义的元类，覆盖了type元类中的__new__()方法和__init__()方法。虽然也可以直接使用TypeCheckMeta(name,bases,dict)这种方式来创建类，但更为常见的是在需要被生成的类中设置__metaclass__属性，两种用法是等价的。

```

class TypeCheck(object):
    __metaclass__ = TypeCheckMeta
    def __setattr__(self, key, value):
        print "set attribute value"
        if key in self._fields:
            if not self._fields[key].is_valid(value):
                raise TypeError('Invalid type for field')
        super(TypeCheck, self).__setattr__(key, value)
class MetaTest(TypeCheck):
    name = TypeSetter(str)
    num = TypeSetter(int)
mt = MetaTest()
mt.name = "apple"
mt.num = "test"

```

当类中设置了__metaclass__属性的时候，所有继承自该类的子类都将使用所设置的元类来指导类的生成，因此上述程序的输出如下：

```

-----
Allocating memory for class TypeCheck
TypeCheck
(<type 'object'>,)
{'__module__': '__main__', '__metaclass__': <class '__main__.TypeCheckMeta'>, '_
__setattr__': <function __setattr__ at 0x00D61830>}
set attribute type <type 'str'>
set attribute type <type 'int'>
-----
Allocating memory for class MetaTest

```

```
MetaTest
(<class '__main__.TypeCheck'>,)
{'__module__': '__main__', 'num': <__main__.TypeSetter object at 0x00D67E70>, 'name': <__main__.TypeSetter object at 0x00D67E50>}
set attribute value
set attribute value
Traceback (most recent call last):
  File "metatest.py", line 38, in <module>
    mt.num = "test"
  File "metatest.py", line 28, in __setattr__
    raise TypeError('Invalid type for field')
TypeError: Invalid type for field
```

实际上，在新式类中当一个类未设置`__metaclass__`属性的时候，它将使用默认的`type`元类来生成类。而当该属性被设置时查找规则如下：

1) 如果存在`dict['__metaclass__']`，则使用对应的值来构建类；否则使用其父类`dict['__metaclass__']`中所指定的元类来构建类，当父类中也不存在指定的`metaclass`的情形下使用默认元类`type`。

2) 对于古典类，条件1不满足的情况下，如果存在全局变量`__metaclass__`，则使用该变量所对应的元类来构建类；否则使用`types.ClassType`。

读者可以通过将上述例子中`__metaclass__=TypeCheckMeta`设置为模块级别或者将`TypeCheck`改为古典类来验证上述查找规则。

需要额外提醒的是，元类中所定义的方法为其所创建的类的类方法，并不属于该类的对象。因此上例中`mt.sayHi()`会抛出`AttributeError: 'MetaTest' object has no attribute 'sayHi'`错误，而调用该方法的正确途径为`MetaTest.sayHi()`。

那么在什么情况下会用到元类呢？有句话是这么说的：当你面临一个问题还在纠结要不要使用元类的时候，往往会有其他的更为简单的解决方案。

Python界的领袖Tim Peters曾这样说过：“元类就是深度的魔法，99%的用户应该根本不必为此操心。如果你想搞清楚究竟是否需要用到元类，那么你就不需要它。那些实际用到元类的人都非常清楚地知道他们需要做什么，而且根本不需要解释为什么要用元类。”

我们来看几个使用元类的场景。

1) 利用元类来实现单例模式。

```
class Singleton(type):
    def __init__(cls, name, bases, dic):
        super(Singleton, cls).__init__(name, bases, dic)
        cls.instance = None
    def __call__(cls, *args, **kwargs):
        if cls.instance is None:
            print "creating a new instance"
            cls.instance = super(Singleton, cls).__call__(
                *args, **kwargs)
        else:
            print "warning:only allowed to create one
                instance,minstance already exists!"
        return cls.instance
class MySingleton(object):
    __metaclass__ = Singleton
```

2) 第二个例子来源于Python的标准库string.Template.string，它提供简单的字符串替换功能。常见的使用例子如下：

```
Template('$name $age').substitute({'name':'admin'}, age=26)
```

该标准库的源代码中就用到了元类，Template的元类为_TemplateMetaclass。_TemplateMetaclass的__init__()方法通过查找属性（pattern、delimiter和idpattern）并将其构建为一个编译好的正则表达式存放在pattern属性中。用户如果需要自定义分隔符（delimiter）可以通过继承Template并覆盖它的类属性delimiter来实现。string.Template的部分源代码如下：

```

class Template:
    """A string class for supporting $-substitutions."""
    __metaclass__ = _TemplateMetaclass
    delimiter = '$'
    idpattern = r'[_a-z][_a-z0-9]*'
    def __init__(self, template):
        self.template = template
class _TemplateMetaclass(type):
    pattern = r"""
    %(delim)s(?:
        (?P<escaped>%(delim)s) |    # Escape sequence of two delimiters
        (?P<named>%(id)s)         |    # delimiter and a Python identifier
        {(?P<braced>%(id)s)}      |    # delimiter and a braced identifier
        (?P<invalid>)             # Other ill-formed delimiter exprs
    )
    """
    def __init__(cls, name, bases, dct):
        super(_TemplateMetaclass, cls).__init__(name, bases, dct)
        if 'pattern' in dct:
            pattern = cls.pattern
        else:
            pattern = _TemplateMetaclass.pattern % {
                'delim' : _re.escape(cls.delimiter),
                'id'    : cls.idpattern,
            }
        cls.pattern = _re.compile(pattern, _re.IGNORECASE | _re.VERBOSE)

```

另外在Django ORM、AOP编程中也有大量使用元类的情形。最后来谈谈关于元类需要注意的几点：

1) 区别类方法与元方法（定义在元类中的方法）。我们先来看一个例子：**Meta**和**SubMeta**都为元类，其中**SubMeta**继承自**Meta**。因此**f1**、**f2**都为元方法，而**Test**为普通类，其元类设置为**SubMeta**，**f3**为类方法。

```

>>> class Meta(type):
...     def f1(cls):
...         print "This is f1()"
...
>>> class SubMeta(Meta):
...     def f2(cls):
...         print "This is f2()"
...
>>>
>>> class Test(object):
...     __metaclass__ = SubMeta
...     @classmethod
...     def f3(cls):
...         print " I am f3()"
...
>>>
>>> t= Test()
>>> SubMeta.f1(Test)
This is f1()

```

```

>>> Meta.f1(Test)
This is f1()
>>> Test.f1()
This is f1()
>>> SubMeta.f2(Test)
This is f2()
>>> Test.f2()
This is f2()
>>> t.f2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Test' object has no attribute 'f2'
>>> Test.f3()
I am f3()
>>> t.f3()
I am f3()
>>>

```

上面的例子说明，元方法可以从元类或者类中调用，而不能从类的实例中调用；但类方法可以从类中调用，也可以从类的实例中调用。

2) 多继承需要严格限制，否则会产生冲突。

```

>>> class M1(type):
...     def __new__(meta, name, bases, atts):
...         print "M1 called for " + name
...         return super(M1, meta).__new__(meta, name, bases, atts)
...
>>> class C1(object):
...     __metaclass__ = M1
...
M1 called for C1
>>>
>>> class Sub1(C1):pass
...
M1 called for Sub1
>>> class M2(type):
...     def __new__(meta, name, bases, atts):
...         print "M2 called for " + name
...         return super(M2, meta).__new__(meta, name, bases, atts)
...
>>> class C2(object):
...     __metaclass__ = M2
...
M2 called for C2
>>> class Sub2(C1, C2):
...     pass
...
M1 called for Sub2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __new__
TypeError: Error when calling the metaclass bases
  metaclass conflict: the metaclass of a derived class must be a (non-strict)
  subclass of the metaclasses of all its bases

```

上面的例子中当Sub2同时继承自元类C1和C2的时候会抛出异常，这是因为Python解释器并不知道C1和C2是否兼容，因此会发出冲突警告。解决冲突的办法是重新定义一个派生自M1和M2的元类，并在C3中将其__metaclass__属性设置为该派生类。

```
>>> class M3(M1, M2):
...     def __new__(meta, name, bases, atts):
...         print "M3 called for " + name
...         return super(M3, meta).__new__(meta, name, bases, atts)
...
>>> class C3(C1, C2):
...     __metaclass__ = M3
...
M3 called for C3
M1 called for C3
M2 called for C3
```



注意

元类用来指导类的生成，元方法可以从元类或者类中调用，不能从类的实例中调用，而类方法既可以从类中调用也可以从类的实例中调用。

建议63：熟悉Python对象协议

因为Python是一门动态语言，Duck Typing的概念遍布其中，所以其中的Concept并不以类型的约束为载体，而另外使用称为协议的概念。所谓协议，类似你讲英语，我也讲英语，我们就可以交流；在Python中就是我需要调用你某个方法，你正好就有这个方法。比如在字符串格式化中，如果有占位符%s，那么按照字符串转换的协议，Python会去自动地调用相应对象的__str__()方法。

```
>>> class Object(object):
...     def __str__(self):
...         print 'called __str__'
...         return super(Object, self).__str__()
...
>>> o = Object()
>>> print "%s" % o
called __str__
<__main__.Object object at 0x10277a5d0>
```

这倒数第二行就是明证。除了__str__()外，还有其他的方法，比如__repr__()、__int__()、__long__()、__float__()、__nonzero__()等，统称类型转换协议。除了类型转换协议之外，还有许多其他协议。

1) 用以比较大小的协议，这个协议依赖于__cmp__()方法，与C语言库函数cmp类似，当两者相等时，返回0，当self<other时返回负值，反之返回正值。因为它的这种复杂性，所以Python又有__eq__()、__ne__()、__lt__()、__gt__()等方法来实现相等、不等、小于和大于的判定。这也就是Python对==、!=、<和>等操作符的进行重载的支撑机制。

2) 数值类型相关的协议，这一类的函数比较多，如表6-2所示。

表6-2 Python的协议与函数对应关系表

分 类	方 法	操作符 / 函数	说 明
数值运算符	<code>__add__</code>	<code>+</code>	加
	<code>__sub__</code>	<code>-</code>	减
	<code>__mul__</code>	<code>*</code>	乘
	<code>__div__</code>	<code>/</code>	除
	<code>__floordiv__</code>	<code>//</code>	整除
	<code>__truediv__</code>	<code>/</code>	真除法，当 <code>__future__.division</code> 起作用时调用，否则调用 <code>__div__</code>
	<code>__pow__</code>	<code>**</code>	幂运算
	<code>__mod__</code>	<code>%</code>	取余
	<code>__divmod__</code>	<code>divmod()</code>	余、除
位运算符	<code>__lshift__</code>	<code><<</code>	向左移位
	<code>__rshift__</code>	<code>>></code>	向右移位
	<code>__and__</code>	<code>&</code>	与
	<code>__or__</code>	<code> </code>	或
	<code>__xor__</code>	<code>^</code>	异或
	<code>__invert__</code>	<code>~</code>	非
运算赋值符	<code>__iadd__</code>	<code>+=</code>	
	<code>__isub__</code>	<code>-=</code>	
	<code>__imul__</code>	<code>*=</code>	
	<code>__idiv__</code>	<code>/=</code>	
	<code>__ifloordiv__</code>	<code>//=</code>	
	<code>__itruediv__</code>	<code>/=</code>	
	<code>__ipow__</code>	<code>**=</code>	
	<code>__imod__</code>	<code>%=</code>	
	<code>__ilshift__</code>	<code><<=</code>	
	<code>__irshift__</code>	<code>>>=</code>	
	<code>__iand__</code>	<code>&=</code>	
	<code>__ior__</code>	<code> =</code>	
	<code>__ixor__</code>	<code>^=</code>	
其他	<code>__pos__</code>	<code>+</code>	正
	<code>__neg__</code>	<code>-</code>	负
	<code>__abs__</code>	<code>abs()</code>	绝对值

基本上，只要实现了表6-2中的几个方法，基本上就能够模拟数值类型了。不过还需要提到一个Python中特有的概念：反运算。别被吓着，其实非常简单。以加法为例，`something+other`，调用的是 `something` 的 `__add__()` 方法，如果 `something` 没有 `__add__()` 方法怎么办呢？调用 `other.__add__()` 是不对的，这时候Python有一个反运算的协

议，它会去查看other有没有__radd__()方法，如果有，则以something为参数调用之。类似__radd__()的方法，所有的数值运算符和位运算符都是支持的，规则也是一律在前面加上前缀r即可，在此不再细表。

3) 容器类型协议。容器的协议是非常浅显的，既然为容器，那么必然要有协议查询内含多少对象，在Python中，就是要支持内置函数len()，通过__len__()来完成，一目了然。而__getitem__()、__setitem__()、__delitem__()则对应读、写和删除，也很好理解。__iter__()实现了迭代器协议，而__reversed__()则提供对内置函数reversed()的支持。容器类型中最有特色的是对成员关系的判断符in和not in的支持，这个方法叫__contains__()，只要支持这个函数就能够使用in和not in运算符了。

4) 可调用对象协议。所谓可调用对象，即类似函数对象，能够让类实例表现得像函数一样，这样就可以让每一个函数调用都有所不同。

```
>>> class Functor(object):
...     def __init__(self, context):
...         self._context = context
...     def __call__(self):
...         print 'do something with %s' % self._context
...
>>> lai_functor = Functor('lai')
>>> yong_functor = Functor('yong')
>>> lai_functor()
do something with lai
>>> yong_functor()
do something with yong
```

5) 与可调用对象差不多的，还有一个可哈希对象，它是通过__hash__()方法来支持hash()这个内置函数的，这在创建自己的类型时非常有用，因为只有支持可哈希协议的类型才能作为dict的键类型（不过只要继承自object的新式类默认就支持了）。

6) 前面的文档谈对描述符协议和属性交互协议 (`__getattr__()`、`__setattr__()`、`__delattr__()`)，那么剩下来还值得一谈的就是上下文管理器协议了，也就是对`with`语句的支持，这个协议通过`__enter__()`和`__exit__()`两个方法来实现对资源的清理，确保资源无论在什么情况下都会正常清理。

```
class Closer:
    '''
    通过with
    语句和一个close
    方法来关闭一个对象'''
    def __init__(self, obj):
        self.obj = obj
    def __enter__(self):
        return self.obj # bound to target
    def __exit__(self, exception_type, exception_val, trace):
        try:
            self.obj.close()
        except AttributeError: # obj isn't closable
            print 'Not closable.'
        return True # exception handled successfully
```

对于实现了这两个方法的`Closer`类，我们可以如下使用它：

```
>>> from ftplib import FTP
>>> with Closer(FTP('ftp.somesite.com')) as conn:
...     conn.dir()
...
>>> conn.dir()
>>>
```

可以看到第二次调用`conn.dir()`已经没有输出，这是因为这个FTP连接会话已被关闭的缘故。与这里`Closer`类似的类在标准库中已经存在，就是`contextlib`里的`closing`。

至此，常用的对象协议就讲完了，只要活学活用这些协议，就能够写出更为Pythonic的代码。不过也要注意，协议不像C++、Java等语言中的接口，它更像是声明，没有语言上的约束力，需要大家共同遵守。

建议64：利用操作符重载实现中缀语法

可能你跟我一样，学完各种对象协议后就跃跃欲试。当年我初学Python的时候，原本最熟悉的编程语言是C++，所以学会操作符重载以后，就拿来炫技了。

```
>>> class endl(object):pass
...
>>> class Cout(object):
...     def __lshift__(self, obj):
...         if obj is endl:
...             print
...             return
...         print obj,
...         return self
...
>>> cout = Cout()
>>> cout << 1 << 2 << endl
1 2
```

如果你像我当年那样初学Python，你就会明白这种模拟C++的流输出让我感觉有多炫！但是现在我知道这是一种对特性的滥用，不应提倡。不过我在这里重提旧事的原因是想要引出一个非常棒的利用操作符重载实现更优雅的代码的例子。

熟悉Shell脚本编程朋友应该都非常熟悉“|”这个符号，它表示管道，用以连接两个应用程序的输入输出。比如按字母表反序遍历当前目录的文件与子目录，可以如下使用：

```
$ ls | sort -r
test
releases
prj
intern
doc
common
branches
art
```

管道的处理非常清晰，因为它是中缀语法。而我们常用的Python是前缀语法的，比如类似的Python代码应该是`sort(ls(),reverse=True)`，明显没有那么清晰，特别是在极限情况下。

```
sum(select(where(take_while(fib(), lambda x: x < 1000000) lambda x: x % 2),
lambda x: x * x))
```

像这样的代码，一堆`sum`、`select`、`where`混在一起，一眼看过去已经头大如斗了。管道符号在Python中，也是或符号，那么有没有可能利用它来简化代码呢？这个想法后来由Julien Palard开发了一个`pipe`库，达成了所愿。这个`pipe`库的核心代码只有几行，就是重载了`__ror__()`方法。

```
class Pipe:
    def __init__(self, function):
        self.function = function
    def __ror__(self, other):
        return self.function(other)
    def __call__(self, *args, **kwargs):
        return Pipe(lambda x: self.function(x, *args, **kwargs))
```

这个`Pipe`类可以当成函数的decorator来使用。比如在列表中筛选数据，可使用如下实现：

```
@Pipe
def where(iterable, predicate):
    return (x for x in iterable if (predicate(x)))
```

`pipe`库内置了一堆这样的处理函数，上文所述的`sum`、`select`、`where`等函数尽在其中，所以马上就可以拿来改造之前的代码。

```
fib() | take_while(lambda x: x < 1000000) \
      | where(lambda x: x % 2) \
      | select(lambda x: x * x) \
      | sum()
```

看，现在是不是一眼就可以看出代码的意义了呢？就是找出小于1000000的斐波那契数，并计算其中的偶数的平方之和。通过这个例子，可以看出中缀语法在这种流式数据处理上的确是非常有优势的。

`pipe`已经发布到pypi，可以使用`pip install pipe`命令安装。安装完成以后可以在Python Shell中尝试一下。

```
>>> from pipe import *
>>> [1, 2, 3, 4, 5] | where(lambda x: x % 2) | tail(2) | select(lambda x: x * x)
| add
34
```

此外，`pipe`是惰性求值的，所以我们完全可以弄一个无穷生成器而不用担心内存被用完。比如：

```
>>> def fib():
...     a, b = 0, 1
...     while True:
...         yield a
...         a, b = b, a + b
```

然后来做一个题目：计算小于4000000的斐波那契数中的偶数之和。

```
>>> euler2 = fib() | where(lambda x: x % 2 == 0) | take_while(lambda x: x <
4000000) | add
>>> assert euler2 == 4613732
```

可以看到代码非常易读，就像读自然语言一样。除了处理数值很方便，用它来处理文本也一样简单。看看这个需求：读取文件，统计文件中每个单词出现的次数，然后按照次数从高到低对单词排序。

```
from __future__ import print_function
from re import split
from pipe import *
with open('test_descriptor.py') as f:
    print(f.read())
```

```
| Pipe(lambda x:split('/W+', x))
| Pipe(lambda x:(i for i in x if i.strip()))
| groupby(lambda x:x)
| select(lambda x:(x[0], (x[1] | count)))
| sort(key=lambda x:x[1], reverse=True)
)
```

看看，非常简单吧？在我这里运行的结果如下：

```
[('self', 13), ('foo', 9), ('item', 9), ('_data', 8), ('print', 7), ('def', 5),
 ('return', 5), ('Jeff', 4), ('i', 4), ('in', 4), ('jeff', 4), ('ken', 4),
 ('obj', 4), ('val', 4), ('class', 3), ('lai', 3), ('pan', 3), ('tmp', 3),
 ('Foo', 2), ('ItemDescriptor', 2), ('Wrapper', 2), ('__iter__', 2), ('for',
2),
 ('if', 2), ('next', 2), ('object', 2), ('0', 1), ('1', 1), ('30', 1), ('8',
1),
 ('None', 1), ('__class__', 1), ('__future__', 1), ('__get__', 1),
('__init__', 1),
 ('__set__', 1), ('bin', 1), ('coding', 1), ('env', 1), ('f', 1), ('from', 1),
 ('import', 1), ('instance', 1), ('isinstance', 1), ('len', 1), ('list', 1),
 ('print_function', 1), ('python', 1), ('type', 1), ('usr', 1), ('utf', 1)]
```

建议65：熟悉 Python 的迭代器协议

其实对于大部分Python程序员而言，迭代器的概念可能并不熟悉。但这很正常，与C++等语言不同，Python的迭代器集成在语言之中，与语言完美地无缝集成，不像C++中那样需要专门去理解这一个概念。比如，要遍历一个容器，Python代码如下：

```
>>> alist = range(2)
>>> for i in alist:
...     print i
0
1
```

而C++代码如下：

```
using namespace std;
vector<int> myIntVector;
//
往容器 myIntVector
中添加元素的操作，略
for(vector<int>::iterator = myIntVector.begin();
    myIntVectorIterator != myIntVector.end();
    myIntVectorIterator++){
    cout<<*myIntVectorIterator<<" ";
}
```

两相对比，可以看到C++的代码中，多了一个`vector<int>::iterator`类型，它是什么、有什么用、什么时候用、怎么用，都是C++程序员需要理解和掌握的内容，所以可以说，在“实现遍历容器”这一事情上，使用C++要付出更多的精力去学习更多的内容，这就是Python把迭代器内建在语言之中的好处。

但是，并非所有的时候都能够隐藏细节，特别是在写一本书向读者讲述其中的机理的时候。所以在这里，首先需要向大家介绍一下`iter()`函数。`iter()`可以输入两个实参，但为了简化起见，在这里忽略第

二个可选参数，只介绍一个参数的形式。`iter()`函数返回一个迭代器对象，接受的参数是一个实现了`__iter__()`方法的容器或迭代器（精确来说，还支持仅有`__getitem__()`方法的容器）。对于容器而言，`__iter__()`方法返回一个迭代器对象，而对迭代器而言，它的`__iter__()`方法返回其自身，所以如果我们用一个迭代器对象`it`，当以它为参数调用`iter(it)`时，返回的是自身。

```
>>> it = iter(alist)
>>> it2 = iter(it)
>>> assert id(it) == id(it2)
```

到时此，就可以跟大家讲一下迭代器协议了。前文已经说过，所谓协议，是一种松散的约定，并没有相应的接口定义，所以把协议简单归纳如下：

- 1) 实现`__iter__()`方法，返回一个迭代器。
- 2) 实现`next()`方法，返回当前的元素，并指向下一个元素的位置，如果当前位置已无元素，则抛出`StopIteration`异常。

可以通过以下代码验证这个协议：

```
>>> alist = range(2)
>>> it = alist.__iter__()
>>> it.next()
0
>>> it.next()
1
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

与上例使用`iter()`内置函数不同，这次的代码是`it=alist.__iter__()`，可见`list`这一容器确实是实现了迭代器协议中容器的部分。后续连续3次的`next()`方法调用也印证了协议的第2条。

熟悉了迭代器协议，那么我们就可以使用它来遍历所有的容器，仍然以list对象为例。

```
>>> alist = range(2)
>>> it = iter(alist)
>>> while True:
...     try:
...         print it.next()
...     except StopIteration:
...         break
...
0
1
```

可以看到输出跟最初使用for循环是一样的，对，你的灵光一闪没有错，其实for语句就是对获取容器的迭代器、调用迭代器的next()方法以及对StopIteration进行处理等流程进行封装的语法糖（类似的语法糖还有in/not in语句）。

迭代器最大的好处是定义了统一的访问容器（或集合）的统一接口，所以程序员可以随时定义自己的迭代器，只要实现了迭代器协议就可以。除此之外，迭代器还有惰性求值的特性，它仅可以在迭代至当前元素时才计算（或读取）该元素的值，在此之前可以不存在，在此之后可以销毁，也就是说不需要在遍历之前事先准备好整个迭代过程中的所有元素，所以非常适合遍历无穷个元素的集合（如斐波那契数列）或巨大的事物（如文件）。

```
class Fib(object):
    def __init__(self):
        self._a = 0
        self._b = 1
    def __iter__(self):
        return self
    def next(self):
        self._a, self._b = self._b, self._a + self._b
        return self._a
for i, f in enumerate(Fib()):
    print f
    if i > 10:
        break
```

这段代码能够打印斐波那契数列的前10项。再来看一下传统的使用容器存储整个数列的方案。

```
>>> def fib(n):
    """
    返回小于指定值的斐波那契数列"""
    result=[]
    a,b=0,1
    while b<n:
        result.append(b)
        a,b=b,a+b
    return result
>>> fib(10)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

与直接使用容器的代码相比，它仅使用两个成员变量，显而易见更省内存，并在一些应用场景更省CPU计算资源，所以在编写代码中应当多多使用迭代器协议，避免劣化代码。对于这一观点，不必怀疑，从Python 2.3版本开始，itertools成为了标准库的一员已经充分印证这个观点。

itertools的目标是提供一系列计算快速、内存高效的函数，这些函数可以单独使用，也可以进行组合，这个模块受到了Haskell等函数式编程语言的启发，所以大量使用itertools模块中的函数的代码，看起来有点像函数式编程语言写推荐，比如sum(imap(operator.mul, vector1,vector2))能够用来运行两个向量的对应元素乘积之和。

itertools最为人所熟知的版本，应该算是zip、map、filter、slice的替代，izip（izip_longest）、imap(startmap)、ifilter(ifilterfalse)、islice，它们与原来的那几个内置函数有一样的功能，只是返回的是迭代器（在Python 3中，新的函数彻底替换掉了旧函数）。

除了对标准函数的替代，itertools还提供以下几个有用的函数：chain()用以同时连续地迭代多个序列；compress()、dropwhile()和takewhile()能用以遴选序列元素；tee()就像同名的UNIX应用程序，对序

列作n次迭代；而groupby的效果类似SQL中相同拼写的关键字所带的效果。

```
[k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
[list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
```

除了这些针对有限元素的迭代帮助函数之外，还有count()、cycle()、repeat()等函数产生无穷序列，这3个函数就分别可以产生算术递增数列、无限重复实参序列的序列和重复产生同一个值的序列。

如果以上这些函数让你感到吃惊，那么接下来的4个组合数学的函数就会让你更加惊讶了，如表6-3所示。

表6-3 组合函数以及其意义

product()	计算 m 个序列的 n 次笛卡尔积
permutations()	产生全排列
combinations()	产生无重复元素的组合
combinations_with_replacement()	产生有重复元素的组合

下面通过例子来熟悉一下，第一行是代码，第二行是结果。

```
product('ABCD', repeat=2)
AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
permutations('ABCD', 2)
AB AC AD BA BC BD CA CB CD DA DB DC
combinations('ABCD', 2)
AB AC AD BC BD CD
combinations_with_replacement('ABCD', 2)
AA AB AC AD BB BC BD CC CD DD
其中 product()
可以接受多个序列，如：
>>> for i in product('ABC', '123', repeat=2): print ''.join(i)
...
A1A1
A1A2
A1A3
A1B1
A1B2
A1B3
#
略去其余输出
```

建议66：熟悉 Python 的生成器

生成器，顾名思义，就是按一定的算法生成一个序列，比如产生自然数序列、斐波那契数列等。之前讲迭代器的时候，就讲过一个生成斐波那契数列的例子。那么迭代器也是生成器？其实不然。迭代器虽然在某些场景表现得像生成器，但它绝非生成器；反而是生成器实现了迭代器协议的，可以在一定程度上看作迭代器。再把话题转回迭代器样式的斐波那契数列实现，熟悉Python的人会觉得其实不简洁，因为还有yield表达式可以简化它。

大概是因为生成器的用处巨大，所以Python中专门有一个关键字来实现它，就是yield。甚至生成器的定义也与这个关键字有关：如果一个函数，使用了yield语句，那么它就是一个生成器函数。当调用生成器函数时，它返回一个迭代器，不过这个迭代器是以生成器对象的形式出现的。所以现在我们来重写一下之前的斐波那契数列实现。

```
def fib(n):
    a, b = 1, 1
    while a < n:
        yield a
        a, b = b, a + b
    for i, f in enumerate(fib(10)):
        print f
```

看，代码行数是不是减少了许多？这就是yield关键字的魅力。不过要掌握这个关键字可不容易，首先来看看fib()函数返回的是什么。

```
>>> f = fib(10)
>>> type(f)
<type 'generator'>
>>> dir(f)
['_class_', '__delattr__', '__doc__', '__format__', '__getattribute__',
 '__hash__', '__init__', '__iter__', '__name__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', 'close', 'gi_code', 'gi_frame', 'gi_running',
 'next', 'send', 'throw']
```

可以看到它返回的是一个**generator**类型的对象，而这个对象带有**__iter__()**和**next()**方法，可见的确是一个迭代器。但那些**next()**、**send()**、**throw()**、**close()**等方法是怎么回事？要理解这些方法，需要我们重温一下手册中的例子。

```
>>> def echo(value=None):
...     print "Execution starts when 'next()' is called for the first time."
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception, e:
...                 value = e
...     finally:
...         print "Don't forget to clean up when 'close()' is called."
...
>>> generator = echo(1)
>>> print generator.next()
Execution starts when 'next()' is called for the first time.
1
```

至此，可以看到每一个生成器函数调用之后，它的函数体并不执行，而是到第一次调用**next()**的时候才开始执行。这一点未免让新手颇为费解，但目前来看除了硬记住这一点外并无它法。要从根源上解决问题的话，可能需要约定生成器函数使用另外一个关键字，比如使用**generator**而不是**def**，不然大家总是会往函数方面去想的。

当第一次调用**next()**方法时，生成器函数开始执行，执行到**yield**表达式为止。如例子中的**value=(yield value)**语句中，只是执行了**yield value**这个表达式，而赋值操作并未执行。记住这一点很重要，只有记住了这一点，才能理解后续的内容，如**send()**方法。

```
>>> print generator.next()
None
```

这个也让人有点困惑，按代码应当是返回1的，怎么返回**None**了呢？这时候需要注意的是代码中的**value=(yield value)**，**yield**是一个表

达式，所以它可以作为一个表达式的右值。当第二次调用`next()`时，`yield`表达式的值赋值给了`value`，而`yield`表达式的默认“返回值”就是`None`，所以后续`value`的值就是`None`。现在再用自然语言来描述一次第二次调用`next()`的过程，首先是`value=(yield value)`语句中的赋值操作得到了执行，即`value`被赋值为`None`，然后是`while`条件判断，再次进入循环体，执行`value=(yield value)`语句，此时`value`的值为`None`，`yield`出来的也是`None`，那么再次调用`next()`时返回`None`就顺理成章了，因为`next()`的返回值就是`yield`表达式的右值。

```
>>> print generator.send(2)
2
```

直率地说，`send()`方法很绕，这不是一个好名字。其实`send()`是全能版本的`next()`，或者说`next()`是`send()`的“快捷方式”，相当于`send(None)`。还记得`yield`表达式有一个“返回值”吗？`send()`方法的作用就是控制这个返回值，使得`yield`表达式的“返回值”是它的实参。

```
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
```

除了能`yield`表达式的“返回值”之外，也可以让它抛出异常，这就是`throw()`方法的能力。在本例中，`yield value`表达式抛出一个`TypeError`异常，然后被内层的`except`语句捕获，并赋值给`value`，因此整个代码的执行流并没有离开`while`循环块，所以进入了下一次循环。当再次执行`yield value`时，异常对象（也就是`value`的值）被返回到此次`throw()`调用中。对于常规业务逻辑的代码来说，处理异常的情况不会像这个例子中那样，而是对特定的异常有很好的处理（比如将异常信息写入日志后优雅地返回），从而实现从外部影响生成器内部的控制流。

```
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

当调用`close()`方法时，`yield`表达式就抛出`GeneratorExit`异常，生成器对象会自行处理这个异常。当调用`close()`之后，再次调用`next()`、`send()`会使生成器对象抛出`StopIteration`异常，换言之，这个生成器对象已经不可再用。最后值得一提的是，当生成器对象被GC回收时，会自动调用`close()`。

除了简化前文中使用迭代器协议生成斐波那契数列的代码之外，生成器还有两个很棒的用处，其中之一是实现`with`语句的上下文管理器协议，利用的是调用生成器函数时函数体并不执行，当第一次调用`next()`方法时才开始执行，并执行到`yield`表达式后中止，直到下一次调用`next()`方法这个特性；其二是实现协程，利用的是上文所述的`send()`、`throw()`、`close()`等特性。在此，继续讲述第一个应用，而第二个应用留待下一小节讲述。

首先，需要我们回过头来重温一下上下文管理器协议，其实就是要求类实现`__enter__()`和`__exit__()`方法。比如以下`file`对象就实现了这个协议：

```
>>> with open('/tmp/xxx.txt', 'w') as f:
...     f.write('hello, context manager.')
... 
```

但是生成器对象并没有这两个方法，所以`contextlib`提供了`contextmanager`函数来适配这两种协议。

```
from contextlib import contextmanager
@contextmanager
def tag(name):
    print "<%s>" % name
    yield
    print "</%s>" % name
>>> with tag("h1"):
...     print "foo"
...
<h1>
```

```
foo
</h1>
```

这是来自Python文档的例子，当进入with块的时候，tag()函数块的第一行执行，并在执行到第二行的时候中止；离开with块的时候，执行print“foo”，完成后执行yield后面的语句，也就是tag()函数块的第三行，然后整个函数执行完毕。通过contextmanager对next()、throw()、close()的封装，yield大大简化了上下文管理器的编程复杂度，对提高代码可维护性有着极大的意义。除了上面这个例子之外，yield和contextmanger也可以用以“池”模式中对资源的管理和回收，具体的实现留给大家去思考。

建议67：基于生成器的协程及greenlet

在前文中，对生成器实现协程卖了个小关子，在这一节，让我们来揭开谜底。不过在此之前，需要先重温一下协程的概念，以及它的意义。

协程，又称微线程和纤程等，据说源于Simula和Modula-2语言，现代编程语言基本上都支持这个特性，比如Lua和ruby都有类似的概念。协程往往实现在语言的运行时库或虚拟机中，操作系统对其存在一无所知，所以又被称为用户空间线程或绿色线程。又因为大部分协程的实现是协作式而非抢占式的，需要用户自己去调度，所以通常无法利用多核，但用来执行协作式多任务非常合适。用协程来做的东西，用线程或进程通常也是一样可以做的，但往往多了许多加锁和通信的操作。下面基于生产者消费者模型，对抢占式多线程编程实现和协程编程实现进行对比。首先来看使用以下线程的实现（伪代码）：

```
//
队列容器
var q := new queue
//
消费者线程
loop
    lock(q)
    get item from q
    unlock(q)
    if item
        use this item
    else
        sleep
//
生产者线程
loop
    create some new items
    lock(q)
    add the items to q
    unlock(q)
```

由以上代码可以看到，线程实现至少有两点硬伤：

·对队列的操作需要有显式/隐式（使用线程安全的队列）的加锁操作。

·消费者线程还要通过sleep把CPU资源适时地“谦让”给生产者线程使用，其中的适时是多久，基本上只能静态地使用经验值，效果往往不尽如人意。

而使用协程可以比较好地解决这个问题。看以下基于协程的生产者消费者模型实现（伪代码）：

```
//
队列容器
var q := new queue
//
生产者协程
loop
    while q is not full
        create some new items
        add the items to q
    yield to consume
//
消费者协程
loop
    while q is not empty
        remove some items from q
        use the items
    yield to produce
```

可以从以上代码看到之前的加锁和谦让CPU的硬伤不复存在，但也损失了利用多核CPU的能力。所以选择线程还是协程，就要看应用场合了。

好，回到主题：协程这东西关Python的生成器什么事？如果你仔细看上面的伪代码，应该留意到其中出现了两个yield！是的，因为yield能够中止当前代码的执行，相当于“让出”CPU资源，跟协程的“协作式”理念不谋而合，所以能够实现协程。

```
def consumer():
    while True:
        line = yield
        print line.upper()
```

```
def producer():
    with open('/var/log/apache2/error_log', 'r') as f:
        for i, line in enumerate(f):
            yield line
            print 'processed line %d' % i
c = consumer()
c.next()
for line in producer():
    c.send(line)
```

依照上文的理念，编写了这些代码，可以看到`consumer()`是一个生成器函数，它接收`yield`表达式的返回值，转换为全大写，并输出到标准输出，然后再次执行`yield`把CPU交给主程序。它的执行结果如下（根据内容会有点不同）：

```
[THU OCT 31 17:49:08 2013] [WARN] INIT: SESSION CACHE IS NOT CONFIGURED [HINT:
    SSLSESSIONCACHE]
processed line 0
HTTPD: COULD NOT RELIABLY DETERMINE THE SERVER'S FULLY QUALIFIED DOMAIN NAME,
    USING APPLTEKIMACBOOK-PRO.LOCAL FOR SERVERNAME
processed line 1
[THU OCT 31 17:49:08 2013] [NOTICE] DIGEST: GENERATING SECRET FOR DIGEST
    AUTHENTICATION ...
processed line 2
[THU OCT 31 17:49:08 2013] [NOTICE] DIGEST: DONE
processed line 3
...
略
```

可以从输出中看到，每输出一行大写的文字后都有一行来自主程序的处理信息，绝不会像抢占式的多线程程序那样“乱序”，这就是协程的“协”字之由来。Python 2.x版本的生成器无法实现所有的协程特性，是因为缺乏对协程之间复杂关系的支持。比如一个`yield`协程依赖另一个`yield`协程，且需要由最外层往最内层进行传值的时候，就没有解决办法。下面就是一个例子：为班级编写一个程序，计算每一个学生的各科总分，并计算班级总分。先尝试编写以下函数：

```
>>> def accumulate():
...     tally = 0
...     while 1:
...         tally += (yield tally)
... 
```

考虑到不同的班级有不同数量的科目，不同的班级有不同数量的学生，所以编写一个生成器进行计算，它可以根据接收到的数值进行计算，无须预先知道数量。现在想象一下你拿到了学生的各科成绩表，可以想象出它是一个二维表，那么代码大概如下：

```
>>> l = []
>>> for s in students:
...     t = 0
...     a = accumulate ()
...     a.next()
...     for c in s:
...         t = a.send(c)
...     l.append(t)
...
>>> t = 0
>>> a = accumulate ()
>>> a.next()
>>> for s in l:
...     t = a.send(s)
...
>>> t
325
```

可以看到无端多出来的对t和a的初始化操作非常刺眼，不过代码总算是可以正常工作。如果你尝试想把它封装成一个用以计算一个学生总分的函数，会更加别扭（想象一下在accumulate()中调用其自身，递归生成器？）。这个问题直到Python 3.3增加了yield from表达式以后才得以解决，通过yield from，外层的生成器在接收到send()或throw()调用时，能够把实参直接传入内层生成器。应用到本例当中，就不需要定义临时容器l来保存每一个学生的成绩，代码复杂性下降许多。下面是假定accumulate使用了yield from后的代码：

```
>>> a = accumulate ()
>>> a.next()
>>> t = 0
>>> for s in students:
...     for klass in s:
...         t += a.send(klass)
...
>>> t
```

看这个嵌套循环的代码是不是简单了许多？回到协程这个主题，因为Python 2.x版本对协程的支持有限，而协程又是非常有用的特性，所以很多Pythonista就开始寻求语言之外的解决方案，并编写了一系列的程序库，其中最受欢迎的莫过于greenlet。

greenlet是一个C语言编写的程序库，它与yield关键字没有密切的关系。greenlet这个库里最为关键的一个类型就是PyGreenlet对象，它是一个C结构体，每一个PyGreenlet都可以看到一个调用栈，从它的入口函数开始，所有的代码都在这个调用栈上运行。它能够随时记录代码运行现场，并随时中止，以及恢复。看到这里，可以发现它跟yield所能够做到的相似，但更好的是它提供从一个PyGreenlet切换到另一个PyGreenlet的机制。最后看一下来自它帮助文件的一个例子，以便对它有个直观的印象。

```
from greenlet import greenlet
def test1():
    print 12
    gr2.switch()
    print 34
def test2():
    print 56
    gr1.switch()
    print 78
gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch()
```

最后一行跳到test1，输出12，跳到test2，输出56，跳回test1，输出34；然后test1执行完，gr1就死了。然后，最初的gr1.switch()调用返回，所以永远也不会输出78。

协程虽然不能充分利用多核，但它跟异步I/O结合起来以后编写I/O密集型应用非常容易，能够在同步的代码表面下实现异步的执行，其中的代表当属将greenlet与libevent/libev结合起来的gevent程序库，它是当下最受欢迎的Python网络编程库。最后，以使用gevent并发查询

DNS的例子作为结束，使用它进行并发查询 n 个域名，能够获得几乎 n 倍的性能提升。

```
>>> import gevent
>>> from gevent import socket
>>> urls = ['www.google.com', 'www.example.com', 'www.python.org']
>>> jobs = [gevent.spawn(socket.gethostbyname, url) for url in urls]
>>> gevent.joinall(jobs, timeout=2)
>>> [job.value for job in jobs]
['74.125.79.106', '208.77.188.166', '82.94.164.162']
```

建议68：理解GIL的局限性

在Python多线程编程中，你有没有遇到过这种问题：多线程Python程序运行的速度比只有一个线程的时候还要慢？除了程序本身的并行性之外，很大程度上与GIL有关。GIL在Python中是一个很有争议的话题，由于它的存在，多线程编程在Python中似乎并不理想，为什么这么说呢？先来了解一下GIL。GIL被称为为全局解释器锁（Global Interpreter Lock），是Python虚拟机上用作互斥线程的一种机制，它的作用是保证任何情况下虚拟机中只会有一个线程被运行，而其他线程都处于等待GIL锁被释放的状态。对于有I/O操作的多线程，其线程执行状态如图6-6所示。不管是在单核系统还是多核系统中，始终只有一个获得了GIL锁的线程在运行，每次遇到I/O操作便会进行GIL锁的释放。

但如果是纯计算的程序，没有I/O操作，解释器则会根据`sys.setcheckinterval`的设置来自动进行线程间的切换，默认情况下每隔100个时钟（注：这里的时钟指的是Python的内部时钟，对应于解释器执行的指令）就会释放GIL锁从而轮换到其他线程的执行，示意图如图6-7所示。

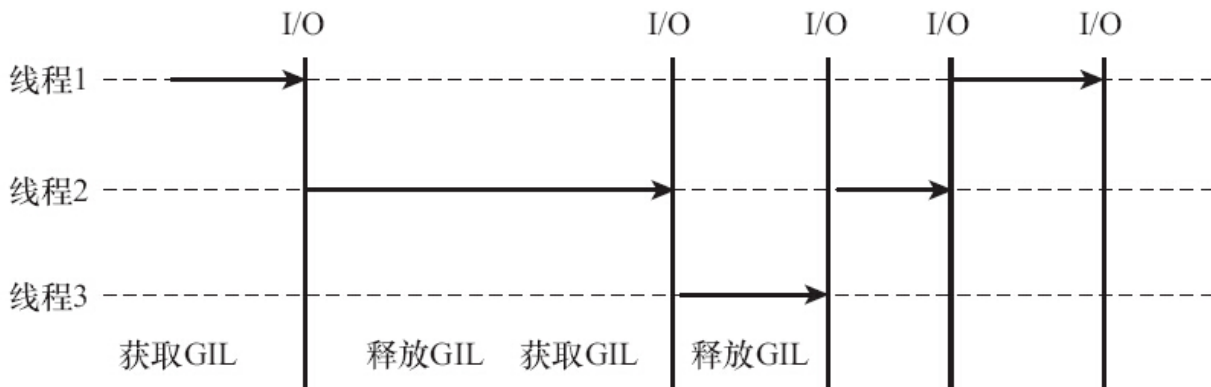


图6-6 Python虚拟机中I/O操作中GIL的变换过程

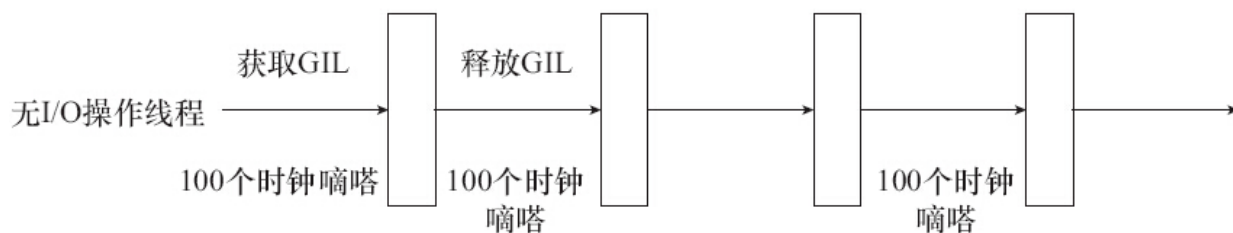


图6-7 无I/O操作时GIL的变换过程

在单核CPU中，GIL对多线程的执行并没有太大影响，因为单核上的多线程本质上就是顺序执行的。但对于多核CPU，多线程并不能真正发挥优势带来效率上明显的提升，甚至在频繁I/O操作的情况下由于存在需要多次释放和申请GIL的情形，效率反而会下降。那么，有人不禁会问：Python解释器中为什么要引入GIL呢？来思考这样一个情形：我们知道Python中对象的管理与引用计数器密切相关，当计数器变为0的时候，该对象便会被垃圾回收器回收。当撤销对一个对象的引用时，Python解释器对对象以及其计数器的管理分为以下两步：

- 1) 使引用计数值减1。
- 2) 判断该计数值是否为0，如果为0，则销毁该对象。

假设线程A和B同时引用同一个对象obj，这时obj的引用计数值为2。如果现在线程A打算撤销对obj的引用。当执行完第一步的时候，由于存在多线程调度机制，A恰好在这个关键点被挂起，而B进入执行状态，如图6-8所示。但不幸的是B也同样做了撤销对obj的引用的动作，并顺利完成了所有两个步骤，这个时候由于obj的引用计数器为0，因此对象被销毁，内存被释放。但如果此时A再次被唤醒去执行第二步操作的时候会发现已经面目全非，则其操作结果完全未知。

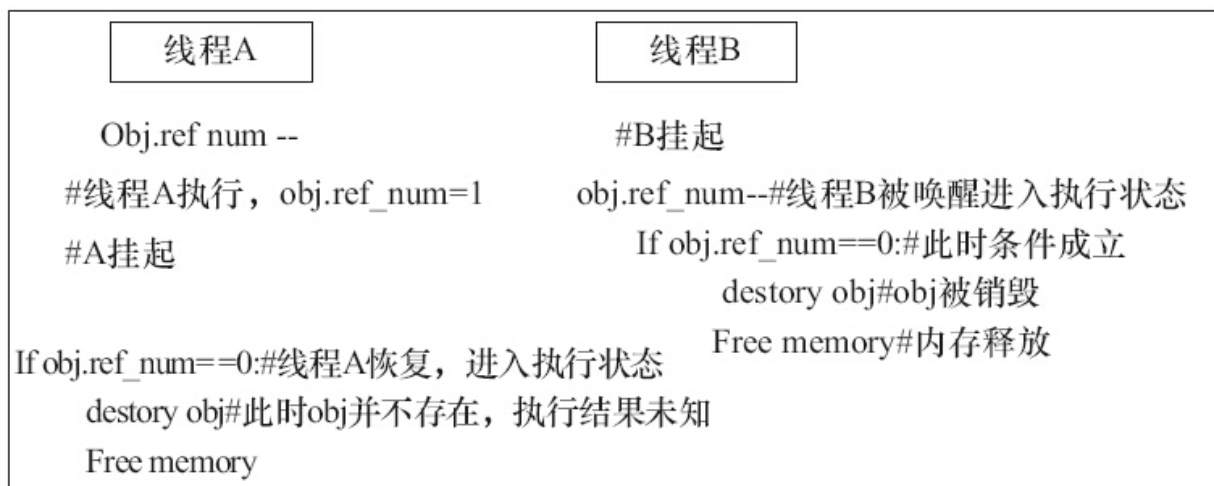


图6-8 无GIL存在时线程的同步

鉴于此，在Python解释器中引入了GIL，以保证对虚拟机内部共享资源访问的互斥性。GIL的引入确实使得多线程不能在多核系统中发挥优势，但它也带来了一些好处：大大简化了Python线程中共享资源的管理，在单核CPU上，由于其本质是顺序执行的，一般情况下多线程能够获得较好的性能。此外，对于扩展的C程序的外部调用，即使其不是线程安全的，但由于GIL的存在，线程会阻塞直到外部调用函数返回，线程安全不再是一个问题。

多核CPU已经成为一个常见的现象，GIL的局限性限制了其在多核CPU上发挥优势，因此对于GIL的去留也曾引发过激烈的讨论。Guido以及Python的开发人员都有一个很明确的解释，那就是去掉GIL并不容易。实际上在1999年，针对Python1.5，Greg Stein发布了一个补丁，该补丁中GIL被完全移除，使用高粒度的锁来代替。然而这种解决方案并没有带来理想的效果，多核多线程速度的提升并没有随着核数的增加而线性增长，反而给单线程程序的执行速度带来了一定的代价，当用单线程执行时，速度大约降低了40%。因此，这种方案最终也被放弃。在Python3.2中重新实现了GIL，其实现机制主要集中在两个方面：一方面是使用固定的时间而不是固定数量的操作指令来进行线程的强制切换；另一个方面是在线程释放GIL后，开始等待，直到某个其他线程获

取GIL后，再开始去尝试获取GIL，这样虽然可以避免此前获得GIL的线程，不会立即再次获取GIL，但仍然无法保证优先级高的线程优先获取GIL。这种方式只能解决部分问题，并未改变GIL的本质，GIL本质上的改观目前并没有非常明朗的前景。不过也不需要那么悲观，Python提供了其他方式可以绕过GIL的局限，比如使用多进程multiprocessing模块或者采用C语言扩展的方式，以及通过ctypes和C动态库来充分利用物理内核的计算能力。

建议69：对象的管理与垃圾回收

通常来说Python并不需要用户自己来管理内存，它与Perl、Ruby等很多动态语言一样具备垃圾回收功能，可以自动管理内存的分配与回收，而不需要编程人员的介入。那么这样是不是意味着用户可以高枕无忧了呢？我们来看下面一个例子：

```
class Leak(object):
    def __init__(self):
        print "object with id %d was born" %id(self)
while(True):
    A = Leak()
    B = Leak()
    A.b = B
    B.a = A
    A = None
    B = None
```

运行上述程序我们会发现，Python进程的内存消耗量一直在持续增长，到最后出现内存耗光的情况。这是什么原因造成的呢？

我们先来简单谈谈Python中内存管理的方式：Python使用引用计数器（Reference counting）的方法来管理内存中的对象，即针对每一个对象维护一个引用计数值来表示该对象当前有多少个引用。当其他对象引用该对象时，其引用计数会增加1，而删除一个对当前对象的引用，其引用计数会减1。只有当引用计数的值为0的时候该对象才会被垃圾收集器回收，因为它表示这个对象不再被其他对象引用，是个不可达对象。引用计数算法最明显的缺点是无法解决循环引用的问题，即两个对象相互引用。上述代码中正是由于形成了A、B对象之间的循环引用而造成了内存泄露的情况，因为两个对象的引用计数器都不为0，该对象并不会被垃圾收集器回收，而无限循环导致一直在申请内存而没有释放，所以最后出现了内存耗光的情况。

循环引用常常会在列表、元组、字典、实例以及函数使用时出现。对于由循环引用而导致的内存泄露的情况，有没有办法进行控制和管理呢？实际上Python自带了一个gc模块，它可以用来跟踪对象的“入引用（incoming reference）”和“出引用（outgoing reference）”，并找出复杂数据结构之间的循环引用，同时回收内存垃圾。有两种方式可以触发垃圾回收：一种是通过显式地调用gc.collect()进行垃圾回收；还有一种是在创建新的对象为其分配内存的时候，检查threshold阈值，当对象的数量超过threshold的时候便自动进行垃圾回收。默认情况下阈值设为(700,10,10)，并且gc的自动回收功能是开启的，这些可以通过gc.isenabled()查看。下面是gc模块使用的简单例子：

```
>>> import gc
>>> print gc.isenabled()
True
>>> gc.isenabled()
True
>>> gc.get_threshold()
(700, 10, 10)
```

对于本节开头的例子，我们使用gc模块来进行垃圾回收，代码如下：

```
def main():
    collected = gc.collect()
    print "Garbage collector before running: collected %d objects." % (collected)
    print "Creating reference cycles..."
    A = Leak()
    B = Leak()
    A.b = B
    B.a = A
    A = None
    B = None
    collected = gc.collect()
    pprint.pprint( gc.garbage)
    print "Garbage collector after running: collected %d objects." % (collected)
if __name__ == "__main__":
    ret = main()
    sys.exit(ret)
```

运行程序输出结果如下：

```
Garbage collector before running: collected 0 objects.  
Creating reference cycles...  
object with id 14109584 was born  
object with id 14109648 was born  
[]  
Garbage collector after running: collected 4 objects.
```

`gc.garbage`返回的是由于循环引用而产生的不可达的垃圾对象的列表，输出为空表示内存中此时不存在垃圾对象。`gc.collect()`显示所有收集和销毁的对象的数目，此处为4（2个对象A、B，以及其实例属性dict）。

我们再来考虑一个问题：如果我们在类**Leak**中添加析构方法`__del__()`，对象的销毁形式和内存回收的情况是否有所不同。示例代码如下：

```
def __del__(self):  
    print "object with id %d was destroyed" %id(self)
```

当加入了析构方法`__del__()`在运行程序的时候会发现`gc.garbage`的输出不再为空，而是对象A、B的内存地址，也就是说这两个对象在内存中仍然以“垃圾”的形式存在。`gc.garbag()`输出如下：

```
[<__main__.Leak object at 0x00D72BF0>, <__main__.Leak object at 0x00D72C30>]
```

这是什么原因造成的呢？实际上当存在循环引用并且当这个环中存在多个析构方法时，垃圾回收器不能确定对象析构的顺序，所以为了安全起见仍然保持这些对象不被销毁。而当环被打破时，`gc`在回收对象的时候便会再次自动调用`__del__()`方法。读者可以自行试验。

`gc`模块同时支持**DEBUG**模式，当设置**DEBUG**模式之后，对于循环引用造成的内存泄露，`gc`并不释放内存，而是输出更为详细的诊断信息为发现内存泄露提供便利，从而方便程序员进行修复。更多`gc`模

块的使用方法读者可以参考文档:

<http://docs.python.org/2/library/gc.html>。

第7章 使用工具辅助项目开发

Python项目的开发过程，其实就是一个或多个包的开发过程，而这个开发过程又由包的安装、管理、测试和发布等多个节点构成，所以这是一个复杂的过程，使用工具进行辅助开发有利于减少流程损耗，提升生产力。本章将介绍几个常用的、先进的工具，比如 `setuptools`、`pip`、`paster`、`nose` 和 `Flask-PyPI-Proxy` 等，这些工具涵盖了项目开发中的几大节点，掌握它们能够让读者在将来的项目开发中达到事半功倍的效果。

建议70：从PyPI安装包

PyPI全称Python Package Index，直译过来就是“Python包索引”，它是Python编程语言的软件仓库，类似Perl的CPAN或Ruby的Gems，目前已经有将近35?000个软件和库（统称为包）提交到上面。既然名字中带有“索引”一词，顾名思义，可以通过包的名字查找、下载、安装PyPI上的包。对于包的作者，在PyPI上注册账号后，还可以登记、更新、上传包等。



注意

PyPI有良好的镜像机制，可以方便地在全球各地架设自有镜像，目前可用的几个镜像列出在PyPI Mirrors页面上，而各个镜像的同步情况可以在专门的网站上看到。因为访问外国网站普遍比较慢，为了方便众多的Python程序员，豆瓣网架设了一个镜像，地址是<http://pypi.douban.com>，访问速度很快，推荐使用（具体的使用方法见easy_install/pip的--index参数）。

某日，你在邮件列表里看到有人推荐requests，写得非常煽情，让你相当有兴趣想要试一试这个号称“更适合给人用”的HTTP客户端库，那么可以打开你的浏览器，并导航到PyPI，在右上角输入reuquests然后单击搜索按钮，如图7-1所示。

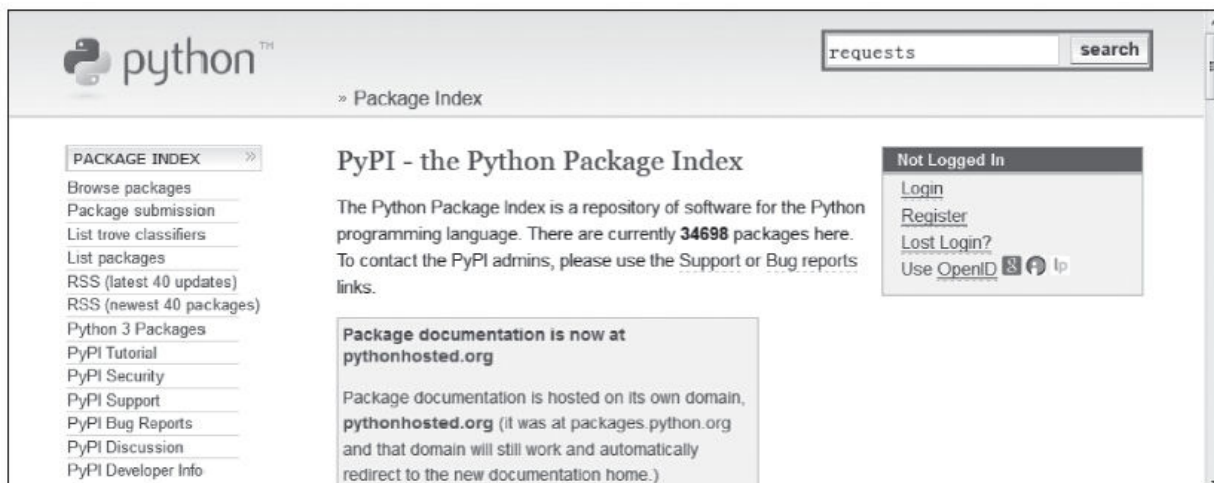


图7-1 PyPI首页

接下来浏览器将会显示搜索结果列表，从中找到名为requests的项目，打开以后页面的上半部分是requests的简要文档，往下拉可以看到它的下载链接，单击后下载保存，然后进入下载目录，执行tar解压缩。

```
~# tar zxvf requests-1.2.3.tar.gz
```

然后进入requests-1.2.3目录安装。

```
~# cd requests-1.2.3
~/requests-1.2.3# python setup.py install
```

Python setup.py install命令将把requests库安装到Python的库目录中。



注意

因为包在PyPI上的主页的URL都是https://pypi.python.org/pypi/{package}的形式，所以在知道包名的情况

下，熟手一般并不使用搜索功能，而是直接手动输入URL。

显然，手动安装包实在是太麻烦了，查找、下载、解压、安装整个流程完全可以自动化。爱好偷懒的Pythonista自然编写好了工具供大家使用，其中`setuptools`尤其值得优先推荐给大家。在Ubuntu Linux上，可以使用`apt`安装这个包。

```
sudo aptitude install python-setuptools
```

其他操作系统大同小异，运行其相应的包管理软件就可安装。但如果你使用MS Windows，则需要去它的主页（<https://pypi.python.org/pypi/setuptools>）下载，然后手动安装。

操作系统对应的软件仓库中的`setuptools`版本通常比较低，所以安装完成以后，最好执行以下命令将其更新到最新版本：

```
easy_install -U setuptools
```

`setuptools`是来自PEAK（Python Enterprise Application Kit，一个致力于提供Python开发企业级应用工具包的项目），由一组发布工具组成，方便程序员下载、构建、安装、升级和卸载Python包，因为它可以自动处理包的依赖关系，所以深受大家的喜爱。



注意

因为PEAK最近几年发展停滞，累及`setuptools`也有好几年没有更新。所以有些程序员重新创建了一个分支项目，称为`distribute`，受到了大家的喜爱。在很长一段时间里，运行`easy_install -U setuptools`更新

的时候，安装的其实是`distribute`。但是，在2013年年初，`distribute`合并到`setuptools`，回归主分支，并发布了`setuptools 0.7`版本。随后几个月频繁发布大版本，至2013年9月，最新的版本已经是1.1.5版，而`distribute`项目也就不再维护了。

安装`setuptools`之后，就可以运行`easy_install`命令了。

```
# easy_install requests
Searching for requests
Reading https://pypi.python.org/simple/requests/
Best match: requests 1.2.3
Downloading
https://pypi.python.org/packages/source/r/requests/requests-
1.2.3.tar.gz#md5=adbd3f18445f7fe5e77f65c502e264fb
Processing requests-1.2.3.tar.gz
Writing /tmp/easy_install-vwjYKV/requests-1.2.3/setup.cfg
Running requests-1.2.3/setup.py -q bdist_egg --dist-dir
/tmp/easy_install-vwjYKV/requests-1.2.3/egg-dist-tmp-MeMFX1
Adding requests 1.2.3 to easy-install.pth file
Installed /usr/local/lib/python2.6/dist-packages/requests-1.2.3-py2.6.egg
Processing dependencies for requests
Finished processing dependencies for requests
```

这就是使用`setuptools`安装`requests`的过程，可以从输出中看到`easy_install`能够查找到最新版本的包，然后进行下载、安装，比手动安装要简单、方便得多。



注意

`setuptools`的功能非常丰富，包括对Python包的构建、测试、发布等都支持得很好，这些功能将在后续的几节中讲述。

建议71：使用pip和yolk安装、管理包

setuptools有几个缺点，比如功能缺失（不能查看已经安装的包、不能删除已经安装的包），也缺乏对git、hg等版本控制系统的原生支持，所以致力于做easy_install改进版的pip在最近几年大受欢迎，成为了最流行的Python包管理工具。

在安装了setuptools以后，安装pip就非常简单了。

```
easy_install pip
```

pip使用子命令形式的CLI接口，首先要学习的当然是help。

```
# pip help
Usage:
  pip <command> [options]
Commands:
  install          Install packages.
  uninstall        Uninstall packages.
  freeze           Output installed packages in requirements format.
  list             List installed packages.
  show             Show information about installed packages.
  search           Search PyPI for packages.
  wheel            Build wheels from your requirements.
  zip              Zip individual packages.
  unzip            Unzip individual packages.
  bundle           Create pybundles.
  help            Show help for commands.
General Options:
  ...
```

从子命令可以对pip的功能有个大体的了解，也可以使用pip help <command>命令查看子命令的帮助信息。Install、uninstall就是用得最多的安装与卸载功能，list可以列出已经安装的包，对感兴趣的包可以使用show命令查看它的具体情况。下面我们重点了解一下这4个命令。

```
# pip install requests
Downloading/unpacking requests
  Downloading requests-1.2.3.tar.gz (348kB): 348kB downloaded
  Running setup.py egg_info for package requests
Installing collected packages: requests
  Running setup.py install for requests
Successfully installed requests
Cleaning up...
```

可以看到pip的install命令使用起来跟easy_install类似，但输出要简洁得多。然后再看看uninstall命令。

```
# pip uninstall requests
Uninstalling requests:
  /usr/local/lib/python2.6/dist-packages/requests-1.2.3-py2.6.egg
Proceed (y/n)? y
  Successfully uninstalled requests
```

删除包时，pip需要输入y进行确认，然后就可以删除干净了。

下面我们来看一段代码：

```
# python
Python 2.6.5 (r265:79063, Oct  1 2012, 22:04:36)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named requests
```

看，找不到名为requests的模块。当然，我们也可以用list命令来确认。

```
# pip list
argparse (1.2.1)
Cheetah (2.0.1)
docutils (0.9)
Flask (0.9)
Flask-RESTful (0.2.1)
FormEncode (1.2.2)
gevent (0.13.0)
GnuPGInterface (0.3.2)
greenlet (0.3.2)
hello-prj (0.0dev)
Jinja2 (2.6)
```

```
...  
# pip list | grep requests
```

安装的包实在太多了，不容易验证是否已删除，若使用`grep`命令帮助查找的结果为空，则`requests`包已经删除干净了。`list`子命令还可以单独列出有新版本的包（使用`--outdated`参数）和已安装最新版本的包（使用`--uptodate`参数）。另外，想要进一步了解比较感兴趣的模块，可以使用`show`命令。比如查看一下`Flask`的信息。

```
# pip show Flask  
---  
Name: Flask  
Version: 0.9  
Location: /usr/local/lib/python2.6/dist-packages/Flask-0.9-py2.6.egg  
Requires: Werkzeug, Jinja2
```

名字、版本、安装路径和依赖信息一样不缺，这对我们了解应用程序的运行环境非常有用。

`pip list`虽然能够列出已经安装的包，但使用上仍然稍有不便，这就是`yolk`存在的意义。`yolk`专注于挖掘已经安装的`Python`包的信息及`PyPI`上的包的信息。

`yolk`与`pip list`功能重复的部分是`yolk {-l|-U}`，`-l`和`-U`这两参数分别用来显示已安装的所有包和可以更新的包，在此就不再介绍了。下面介绍的是`pip`还不具备的功能。

```
# yolk --entry-map nose  
{'console_scripts': {'nosetests': EntryPoint.parse('nosetests = nose:  
    run_exit'),  
    'nosetests-2.6': EntryPoint.parse('nosetests-2.6 =  
    nose:run_exit')},  
'distutils.commands': {'nosetests': EntryPoint.parse('nosetests = nose.  
    commands:nosetests')}}}
```

`yolk--entry-map <package>`可以显示包注册的所有入口点，这样可以了解到安装的包都提供了哪些命令行工具，或者支持哪些基于entry-point的插件系统。上例中，`nose`提供了`nosetests`命令并实现了`distutils.commands`插件（即`python setup.py nosetests`扩展命令）。那么如何查看有哪些包实现了某一个包的插件协议呢？`--entry-points`参数可以帮到你。

```
# yolk --entry-points abu.admin
read_and_display.admin
    read_and_display = read_and_display.admin:Admin
caihui.bs.admin
    caihui.bs = caihui.bs.admin:Admin
```

可以看到，支持`abu.admin`插件协议的包有两个：`read_and_display`和`caihui.bs`，它们的入口点名字分别是`read_and_display.admin`和`caihui.bs.admin`。怎么样，是不是对自己机器上安装了的包有了更深刻的理解？

最后，如果你使用的是桌面版的操作系统，利用`yolk-H<package>`可以打开一个浏览器，并将你指定的包显示在PyPI上的主页，从此告别手动拼接URL的历史。当然，`yolk`还有更多的功能可通过阅读它的手册逐步发掘。

建议72：做paster创建包

如果有一个小程序，或者很简单一个库，举个例子，假定编写了一个四则运算的库。

```
def add(x, y):  
    return x + y  
def division(x, y):  
    return x / y  
def multiply(x, y):  
    return x * y  
def subtract(x, y):  
    return x - y
```

可以把代码保存到一个名为`arithmetic.py`的文件中，然后复制到需要的文件目录中以备使用。比如在一个简单的计算项目中，我们可以这样使用刚编好的库：

```
import arithmetic  
print arithmetic.add(5, 8)  
print arithmetic.subtract(10, 5)  
print arithmetic.division(2, 7)  
print arithmetic.multiply(12, 6)
```

如果只是个人项目，或者很小的团队协作，这种做法问题不大。但如果团队比较大，就有几个问题：

1) 程序的发布。如果版本更新了，如何快速地发布给团队中的所有人。

2) 保证版本同步。`arithmetic.py`不带有任何版本信息，不利于团队成员自检版本。

显然，这个四则运算程序库最好是像之前讨论过的requests之类的优秀的第三方库一样，可以方便地下载、安装、升级、卸载，也就是说能够放到PyPI上面，也能够很好地跟pip或yolk这样的工具集成。Python作为“自带电池”的高级语言，自然提供了这方面的支持，那就是distutils标准库。distutils标准库至少提供了以下几方面内容：

- 1) 支持包的构建、安装、发布（打包）。
- 2) 支持PyPI的登记、上传。
- 3) 定义了扩展指令的协议，包括distutils.cmd.Command基类、distutils.commands和distutils.key_words等入口点，为setuptools和pip等提供了基础设施。

要使用distutils，按习惯需要编写一个setup.py文件，作为后续操作的入口点。在arithmetic.py同层目录，建立一个setup.py文件。

```
# tree .
.
├── arithmetic.py
└── setup.py
```

它的内容如下：

```
from distutils.core import setup
setup(name='arithmetic',
      version='1.0',
      py_modules=['arithmetic'],
      )
```

一眼就可以看出，setup.py文件的意义是执行时调用distutils.core.setup()函数，而实参是通过命名参数指定的。name参数指定的是包名；version指定版本；而py_modules参数是一个序列类型，里面包含需要安装的Python文件，在本例中即为arithmetic.py。

编写好setup.py文件以后，就可以使用python setup.py install把它安装到系统中了。

```
# python setup.py install
running install
running build
running build_py
creating build
creating build/lib.linux-x86_64-2.6
copying arithmetic.py -> build/lib.linux-x86_64-2.6
running install_lib
copying build/lib.linux-x86_64-2.6/arithmetic.py -> /usr/local/lib/python2.6/
dist-packages
byte-compiling /usr/local/lib/python2.6/dist-packages/arithmetic.py to
arithmetic.pyc
running install_egg_info
Writing /usr/local/lib/python2.6/dist-packages/arithmetic-1.0.egg-info
```

安装成功后，我们来试一下。

```
# python
>>> import arithmetic
>>> dir(arithmetic)
['_builtins_', '__doc__', '__file__', '__name__', '__package__', 'add',
'division', 'multiply', 'subtract']
>>> arithmetic.add(1, 2)
3
```

完全符合预期啊！接下来再用yolk查看一下。

```
# yolk -l | grep arithmetic
arithmetic      - 1.0          - active development (/usr/local/lib/python2.6/
dist-packages)
```

可以看到它的确跟其他的Python一样被安排到了系统当中。除了install命令以外，distutils还带有其他命令，可以通过python setup.py--help-commands进行查询。

```
# python setup.py --help-commands
Standard commands:
  build          build everything needed to install
  build_py       "build" pure Python modules (copy to build directory)
  build_ext      build C/C++ extensions (compile/link to build directory)
  build_clib     build C/C++ libraries used by Python extensions
```

```
build_scripts    "build" scripts (copy and fixup #! line)
clean            clean up temporary files from 'build' command
install         install everything from build directory
install_lib      install all Python modules (extensions and pure Python)
install_headers  install C/C++ header files
install_scripts  install scripts (Python or otherwise)
install_data     install data files
sdist           create a source distribution (tarball, zip file, etc.)
register         register the distribution with the Python package index
bdist           create a built (binary) distribution
bdist_dumb       create a "dumb" built distribution
bdist_rpm        create an RPM distribution
bdist_wininst    create an executable installer for MS Windows
upload          upload binary package to PyPI
usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
       or: setup.py --help [cmd1 cmd2 ...]
       or: setup.py --help-commands
       or: setup.py cmd --help
```

在这里，就只讲述`install`指令，其他指令，比如`sdist`、`register`、`upload`将在建议78中讲述，而其他更多指令，请参考`distutils`的文档。

`arithmetic`只是一个示例性的小项目，所以`setup.py`文件非常简单。实际上若要把包提交到PyPI，还需要遵循PEP 241，给出足够多的元数据才行，比如对包的简短描述、详细描述、作者、作者邮箱、主页和授权方式等。比如`requests`的`setup.py`文件中调用`setup()`函数时就包含以下内容：

```
setup(
    name='requests',
    version=requests.__version__,
    description='Python HTTP for Humans.',
    long_description=open('README.rst').read() + '\n\n' +
                    open('HISTORY.rst').read(),
    author='Kenneth Reitz',
    author_email='me@kennethreitz.com',
    url='http://python-requests.org',
    packages=packages,
    package_data={'': ['LICENSE', 'NOTICE'], 'requests': ['*.pem']},
    package_dir={'requests': 'requests'},
    include_package_data=True,
    install_requires=requires,
    license=open('LICENSE').read(),
    zip_safe=False,
    classifiers=(
        'Development Status :: 5 - Production/Stable',
        'Intended Audience :: Developers',
        'Natural Language :: English',
        'License :: OSI Approved :: Apache Software License',
        'Programming Language :: Python',
        'Programming Language :: Python :: 2.6',
        'Programming Language :: Python :: 2.7',
        'Programming Language :: Python :: 3',
    )
)
```

```
        'Programming Language :: Python :: 3.3',  
    ),  
)
```

包含太多内容了，如果每一个项目都手写，肯定记不清楚，所以最好找一个工具可以自动创建项目的`setup.py`文件以及相关的配置、目录等。Python中做这种事的工具有好几个，但做得最好的是`pastescript`。`pastescript`是一个有着良好插件机制的命令行工具，安装以后就可以使用`paster`命令，创建适用于`setuptools`的包文件结构。

首先需要安装`pastescript`。

```
pip install pastescript
```

安装以后可以看到它注册了一个命令行入口`paster`（还有许多插件协议的实现，关于插件协议实现的具体内容不属于本节讨论范围，因此此处不做深入探讨）。

```
# yolk --entry-map pastescript  
{'console_scripts': {'paster': EntryPoint.parse('paster = paste.script.  
    command:run')}},  
...
```

接下来学习一下`paster`怎么用（因为本节专注于讲述包的创建，所以略去了无关的输出）。

```
# paster --help  
Usage: paster [paster_options] COMMAND [command_options]...  
Commands:  
  create      Create the file layout for a Python distribution  
  help        Display help  
...
```

`paster`支持多个命令，现在首先要学习的就是`create`。`create`命令用于根据模板创建一个Python包项目，创建的项目文件结构可以使用

setuptools直接构建，并且支持setuptools扩展的distutils命令，如develop命令等。接下来看一下它的帮助（略去与本节无关的参数）。

```
# paster help create
Usage: /usr/bin/paster create [options] PACKAGE_NAME [VAR=VALUE VAR2=VALUE2 ...]
...
Options:
...
  -t TEMPLATE, --template=TEMPLATE
                                Add a template to the create process
...
  --list-templates              List all templates available
...
  --config=CONFIG              Template variables file
```

显然，`--template`参数用以指定使用的模板。那么又有哪些模板可以使用呢？这就需要先用`--list-templates`查询一下目前安装的模板了。

```
# paster create --list-template
Available templates:
  basic_package:      A basic setuptools-enabled package
  paste_deploy:       A web application deployed through paste.deploy
```

如果一个全新的环境只安装好了paster，那么一般只有两个模板：`basic_package`和`paste_deploy`，在这一节，只需要关注前者。

```
# paster create -o arithmetic-2 -t basic_package arithmetic
Selected and implied templates:
  PasteScript#basic_package  A basic setuptools-enabled package
Variables:
  egg:      arithmetic
  package:  arithmetic
  project:  arithmetic
Enter version (Version (like 0.1)) ['']: 0.0.1
Enter description (One-line description of the package) ['']: first paster prj
Enter long_description (Multi-line description (in reST)) ['']: first paster
prj long sdescriptoin
Enter keywords (Space-separated keywords/tags) ['']:
Enter author (Author name) ['']: lyh
Enter author_email (Author email) ['']: lyh@example.com
Enter url (URL of homepage) ['']: http://code.example.com/arithmetic
Enter license_name (License name) ['']: MIT
Enter zip_safe (True/False: if the package can be distributed as a .zip file)
[False]:
Creating template basic_package
Creating directory arithmetic-2/arithmetic
...
Running /usr/bin/python setup.py egg_info
```

可以看到，简单地填写几个问题以后，paster就在arithmetic-2目录生成了名为arithmetic的包项目。它的文件结构如下：

```
# tree arithmetic-2/
arithmetic-2/
├── arithmetic
│   └── arithmetic
├── __init__.py
├── arithmetic.egg-info
├── dependency_links.txt
├── entry_points.txt
├── not-zip-safe
├── PKG-INFO
├── SOURCES.txt
├── top_level.txt
├── docs
├── license.txt
├── setup.cfg
└── setup.py
```

然后再看一下我们最为关注的setup.py文件。

```
# cat arithmetic-2/arithmetic/setup.py
from setuptools import setup, find_packages
import sys, os
version = '0.0.1'
setup(name='arithmetic',
      version=version,
      description="first paster prj",
      long_description="""\
first paster prj long sdescriptoin""",
      classifiers=[], # Get strings from http://pypi.python.org/pypi? % 3
                    Aaction=list_classifiers
      keywords='',
      author='lyh',
      author_email='lyh@example.com',
      url='http://code.example.com/arithmetic',
      license='MIT',
      packages=find_packages(exclude=['ez_setup', 'examples', 'tests']),
      include_package_data=True,
      zip_safe=False,
      install_requires=[
          # -*- Extra requirements: -*-
      ],
      entry_points="""
      # -*- Entry points: -*-
      """,
      )
```

看，所有的参数都帮我们填好了。是不是有一种“有了paster，再也不担心创建包项目了”的感觉呢？不过如果是第一次使用paster的话，回答问题时可能会输入错误，而paster又不能回退删除输入，所以一出错就只好重来一次。另外，如果创建的是公司项目，那么很多参数的值都是确定的，比如author的值一般就是公司名，那么每次都要重新输入也非常麻烦。要解决上述的两个问题，就可以用上--config参数了，它是一个类似ini文件格式的配置文件，可以使用你喜欢的编辑器在里面填好各个模板变量的值（查询模板有哪些变量用--list-variables参数），然后就可以使用了。比如我们编辑了公司项目专用的模板文件corp-prj-setup.cfg:

```
[pastescript]
description = corp-prj
license_name =
keywords = Python
long_description = corp-prj
author = xxx corp
author_email = xxx@example.com
url = http://example.com
version = 0.0.1
```

然后这样使用:

```
paster create -t basic_package --config="corp-prj-setup.cfg" arithmetic
```

paster将不再向你询问，而是直接生成整个项目，这进一步减轻了创建Python包的工作量。因为使用paster确实能够帮助开发人员减轻创建项目的工作量，所以受到了许多项目的青睐，比如Pyramid就带有几个模板，能够通过模板快速地创建基于Pyramid的Web项目。

建议73：理解单元测试概念

单元测试用来验证程序单元的正确性，一般由开发人员完成，是测试过程的第一个环节，以确保所写的代码符合软件需求和遵循开发目标。好的单元测试可以带来以下好处：

- 减少了潜在bug，提高了代码的质量。经过了严格的单元测试，意味着代码中潜在的bug数量大大减少，同时单元测试能够使得你的思考方式不同于编码，因此能够很快发现不合理的设计或者逻辑，以及算法方面的漏洞或者故障，从而为编写高质量代码提供基本保障。

- 大大缩减软件修复的成本。我们知道在软件开发生命周期越早阶段发现问题或缺陷，其修复的代价越小，因此在单元测试阶段发现问题后修复代价要远远小于在集成测试或者系统测试阶段的代价。

- 为集成测试提供基本保障。单元测试可以大大减少程序中各个部件的不可靠性，通过先测试程序部件再测试部件组装，使集成测试变得更加简单，测试人员因此可以将更多的精力放在用户场景上。

纵然单元测试有各种好处，事实却往往是“理想很丰满，现实很骨感”。实际应用中，单元测试的实践并不理想，原因是多方面的：一则管理层重视不够，根本没有把单元测试提升到和系统集成测试同样的高度；二则是迫于项目期限的压力，开发人员往往没有更多的时间来写单元测试的用例和代码；三则开发人员本身存有趋利避害的侥幸心理，他们更关注于可以工作的代码，一旦编码完成，便迫切地希望能够进行集成工作，因为这样进度看起来更快，同时寄希望于集成测试去发现程序中潜在的问题。那么，到底应该怎么样进行有效的单元测试呢？有效的单元测试应该从以下几个方面考虑：

1) 测试先行，遵循单元测试步骤。测试不应该是编码结束后再来考虑的事情，实际上从项目需求阶段就应该开始考虑。编写单元测试应该尽量安排在项目的早期，并且测试代码应该先于被测试的代码，这样更有利于明确需求。典型的单元测试的步骤如下：

- 创建测试计划（Test Plan）。
- 编写测试用例，准备测试数据。
- 编写测试脚本。
- 编写被测代码，在代码完成之后执行测试脚本。
- 修正代码缺陷，重新测试直到代码可接受为止。

2) 遵循单元测试基本原则。常见的原则如下：

·**一致性**。意味着1000次执行和一次执行的结果应该是一样的，因此，类似于`currenttime=time.localtime()`，产生这种不确定执行结果的语句应该尽量避免。

·**原子性**。意味着单元测试的执行结果返回只有两种，**True**或者**False**，不存在部分成功、部分失败的例子。如果发生这样的情况，往往是测试设计得不够合理。

·**单一职责**。测试应该基于情景（**scenario**）和行为，而不是方法。如果一个方法对应着多种行为，应该有多个测试用例；而一个行为即使对应多个方法也只能有一个测试用例。例如下边的代码。

```
testMethod():  
    assertTrue(behaviour1)  
    assertTrue(behaviour2)
```

应该改为:

```
testMethodCheckBehaviour1
() :
    assertTrue(behaviour1)
testMethodCheckBehaviour2
() :
    assertTrue(behaviour2)
```

·**隔离性**。不能依赖于具体的环境设置，如数据库的访问、环境变量的设置、系统的时间等；也不能依赖于其他的测试用例以及测试执行的顺序，并且无条件逻辑依赖。单元测试所有的输入应该是确定的，方法的行为和结果应是可以预测的。因此要避免以下的测试例子:

```
testMehodBeforeOrAfter():
    if before:
        assertTrue(behaviour1)
    elif after:
        assertTrue(behaviour2)
    else:
        assertTrue(behaviour3)
```

修改为:

```
testMethodBefore():
    before = True
    assertTrue(behaviour1)
testMethodAfter():
    after= True
    assertTrue(behaviour2)
testMethodNow():
    after= False
    before = False
    assertTrue(behaviour3)
```

3) 使用单元测试框架。Python测试也曾经历过“蛮荒时代”，那个时候测试完全是个人化的行为，没有统一的框架标准，每个用Python构建的项目在编写和运行测试方面都采用自己的习惯做法。这种做法不仅效率低下，而且不利于项目管理。幸好后来Python社区出现了一

些测试套件，提供约定和通用标准，后面逐渐演变为流行的测试框架。在单元测试方面常见的测试框架有PyUnit等，它是Kent Beck和Erich Gamma所设计的JUnit的Python版本，在Python 2.1之前需要单独安装，Python 2.1之后它成为一个标准库，名为unittest。它支持单元测试自动化，可以共享地进行测试环境的设置和清理，支持测试用例的聚集以及独立的测试报告框架。我们以unittest来看看如何借助单元测试框架更好地进行单元测试。unittest相关的概念主要有以下4个：

- 测试固件（test fixtures）。测试相关的准备工作和清理工作，基于类TestCase创建测试固件的时候通常需要重新实现setUp()和tearDown()方法。当定义了这些方法的时候，测试运行器会在运行测试之前和之后分别调用这两个方法。

- 测试用例（test case）。最小的测试单元，通常基于TestCase构建。

- 测试用例集（test suite）。测试用例的集合，使用TestSuite类来实现，除了可以包含TestCase外，也可以包含其他TestSuite。

- 测试运行器（test runner）。控制和驱动整个单元测试过程，一般使用TestRunner类作为测试用例的基本执行环境，常用的运行器为TextTestRunner，它是TestRunner的子类，以文字方式运行测试并报告结果。

来看一个简单实例。假设要测试下述类：

```
class MyCal(object):
    def add(self,a,b):
        return a+b
    def sub(self,a,b):
        return a-b
```

首先编写测试用例，并在setUp()方法中完成初始化工作，在tearDown()方法中完成资源释放相关的工作。我们采用动态方法编写测试类，多个测试方法可以集成在一个类中，这些方法按习惯通常以test开头。具体如下：

```
class MyCalTest(unittest.TestCase):
    def setUp(self):
        print "running set up"
        self.mycal = mycal.MyCal()
    def tearDown(self):
        print "running teardown"
        self.mycal = None
    def testAdd(self):
        self.assertEqual(self.mycal.add(-1,7), 6)
    def testSub(self):
        self.assertEqual(self.mycal.sub(10,2), 8)
```

在创建了一些TestCase子类的实例作为测试用例之后，下一步要做的工作就是用TestSuite类来组织它们。TestSuite类可以看成是TestCase类的一个容器，用来对多个测试用例进行组织，这样多个测试用例可以自动在一次测试中全部完成。

```
suite = unittest.TestSuite()
suite.addTest(MyCalTest("testAdd"))
suite.addTest(MyCalTest("testSub"))
```

在编写完测试用例及组织好测试用例之后，现在可以执行测试了。

```
runner = unittest.TextTestRunner()
runner.run(suite)
```

运行命令python-m unittest-v MyCalTest，得到测试结果的输出如下：

```
testAdd (MyCalTest.MyCalTest) ... running set up
running teardown
ok
```

```
testSub (MyCalTest.MyCalTest) ... running set up
running teardown
ok
-----
Ran 2 tests in 0.000s
OK
```

当然实际执行测试过程和应用测试框架比上面的例子要复杂得多。读者可以在Python文档中查看更多关于unittest使用的详细信息，并在实际工作中实践。

最后需要强调的是，单元测试绝不是浪费时间的无用功，它是高质量代码的保障之一，在软件开发的环节中值得投入精力和时间把好这一关。

建议74：为包编写单元测试

当我们创建了一个包，接着就开始为它编写业务逻辑的代码。比如在前文中，我们创建了`arithmetic`包，并在里面增加一个加法函数，如下：

```
def add(x, y):  
    return x + y
```

无名氏说：“当你写下代码，`bug`随之而来”。所以我们需要对代码进行测试，以便交付物在交付给业务的下游部门使用时有一定质量保障。对于一个函数而言，最简单的方法也许就是为它编写一些单元测试代码了。

```
if __name__ == "__main__":  
    assert add(1, 2) == 3  
    assert add(1, -1) == 0
```

这样，当以`arithmetic.py`为入口文件执行`arithmetic.py`的时候，就会运行这些测试代码，实现对`add()`函数的质量检测。像这种针对函数编写的测试，我们称为“单元测试”，它是白盒测试的一种，所以单元测试用例都是根据函数的代码而制定的。通过单元测试，可以有效地避免软件退化，增进软件质量，并更快地产生健壮的代码。甚至对开发人员来说，单元测试用例也是最好的文档。

虽然上例让大家感觉测试非常简单，但实际项目中的测试也有不少麻烦：

1) 程序员希望测试更加自动化，想象一下，如果加减乘除4个函数不是实现在`arithmetic.py`一个文件中，而是分列在4个文件中，那么要测试它们就需要分别运行这4个文件。再想象一下，实际项目中可能一个包中有几十甚至上百个文件，那么想要全部测试一次就非常困难。

2) 一个测试用例往往在测试之前需要进行打桩或做一些准备工作，在测试之后要清理现场，最好有一个框架可以自动完成这些工作。

3) 对于大项目，大量的测试用例需要分门别类地放置，而测试之后，分别产生相应的测试报告。

Python是一门务实的语言，所以自带的电池中就包含了一个名为`unittest`的模块，可以解决这些问题。关于`unittest`的知识，我们在建议73中已经学过，接下来就看一下如何使用`unittest`进行测试的代码。

```
import unittest
import arithmetic
class TestCase(unittest.TestCase):
    def test_add(self):
        self.assertEqual(arithmetic.add(1, 1), 2)
if __name__ == "__main__":
    unittest.main()
把这些代码保存到 test_arithmetic.py
中，然后执行命令：
>python test_arithmetic.py
.
-----
Ran 1 test in 0.000s
OK
```

虽然没有显式地调用`TestCase.test_add`，但从`Ran 1 test in 0.000s`这句输出中可以看到这个测试用例已经执行到了，这就是框架的好处。除了自动调用匹配以`test`开头的方法之外，`unittest.TestCase`还有模板方法`setUp()`和`tearDown()`，通过覆盖这两个方法，能够实现在测试之前执行一些准备工作，并在测试之后清理现场。

然后再回到最初的假设：在arithmetic项目中，若加减乘除4个函数分别在不同的文件中，那么测试用例也可能分别写在4个文件中，那么运行python test_xxx.py命令的形式就无法简化测试工作。这时候可以使用unittest的测试发现（test discover）功能。

```
>python -m unittest discover
.
-----
Ran 1 test in 0.000s
OK
```

unittest将递归地查找当前目录下匹配test*.py模式的文件，并将其中的unittest.TestCase的所有子类都实例化，然后调用相应的测试方法进行测试，这就一举解决了“通过一条命令运行全部测试用例”的问题。

unittest的测试发现功能是Python 2.7版本中才有的，如果你在使用更旧的版本，请安装unittest2。

尽管unittest的测试发现功能已经非常方便，但是因为它需要高版本的Python支持，所以大家喜欢使用setuptools的扩展命令test。

```
>python setup.py test
running test
running egg_info
writing arithmetic.egg-info/PKG-INFO
writing top-level names to arithmetic.egg-info/top_level.txt
writing dependency_links to arithmetic.egg-info/dependency_links.txt
writing entry points to arithmetic.egg-info/entry_points.txt
reading manifest file 'arithmetic.egg-info/SOURCES.txt'
writing manifest file 'arithmetic.egg-info/SOURCES.txt'
running build_ext
test_add (test_arithmetic.TestCase) ... ok
-----
Ran 1 test in 0.000s
OK
```

setuptools对distutils.commands进行了扩展，增加了test命令。如上例所示，这个命令执行的时候，先运行egg_info和build_ext子命令构建项目，然后把项目路径加到sys.path中，再搜寻所有的测试套件（test

suite，通常指多个测试用例或测试套件的组合），并运行之。要使用这个扩展命令，需要在调用setup()函数的时候向它传递test_suite元数据。比如在arithmetic项目中，是这样的：

```
>cat setup.py
...
setup(name='arithmetic',
...
      test_suite = "test_arithmetic",
...

```

test_suite元数据的值可以指向一个包、模块、类或函数，比如在著名的flask项目中，是test_suite='flask.testsuite.suite'，其中flask.testsuite.suite是一个函数；而在arithmetic项目中，test_arithmetic是一个模块。

使用setuptools的测试发现功能，可以给开发人员更一致的开发体验，就像使用build、install命令一样，所以受到了大家的喜爱。但是来自unittest本身的缺陷让开发人员想要找到一个更好的测试框架。

- 1) unittest并不够Pythonic，比如从JUnit中继承而来的首字母小写的骆驼命令法；所有的测试用例都需要从TestCase继承。
- 2) unittest的setUp()和tearDown()只是在TestCase的层面上提供，即每一个测试用例执行的时候都会运行一遍，如果有许多模块需要测试，那么创建环境和清理现场操作都会带来大量工作。
- 3) unittest没有插件机制进行功能扩展，比如想要增加测试覆盖统计特性就非常困难。

nose就是作为更好的测试框架进入大家视线的，而且它更是一个具有更强大的测试发现运行的程序。此外nose定义了插件机制，使得扩展nose的功能成为可能（默认自带coverage插件）。使用pip、

easy_install安装以后，就多了一个nosetests命令可以使用。比如在arithmetic项目中运行：

```
>nosetests -v
test_add (test_arithmetic.TestCase) ... ok
-----
Ran 1 test in 0.004s
OK
```

可以看到nose能够自动发现测试用例，并调用执行，由于它与原有的unittest测试用例兼容，所以可以随时将它引入到项目中来。其实nose的测试发现机制更进一步，它抛弃了unittest中测试用例必须放在TestCase子类中的限制，只要命名符合(?:^|[b_.-])[Tt]est正则表达式的类和函数都可作为测试用例运行。

此外，nose作为一个测试框架，也提供了与unittest.TestCase类似的断言函数，但它抛弃了unittest的那种Java风格的命令方式，使用的是符合PEP8的命名方式。

针对unittest中setUp()和tearDown()只能放在TestCase中的问题，nose提供了3个级别的解决方案，这些配置和清理函数，可以放在包（__init__.py文件中）、模块和测试用例中，非常完美地解决了不同层次的测试需要的配置和清理需求。

最后，nose与setuptools的集成更加友好，提供了nose.collector作为通过的测试套件，让开发人员无须针对不同项目编写不同的套件。比如针对arithmetic项目的setup.py文件作如下修改：

```
>cat setup.py
...
setup(name='arithmetic',
...
#       test_suite = "test_arithmetic",
      test_suite = "nose.collector",
...

```

然后运行`python setup.py test`，得到的结果是一样的。因为使用了`nose.collector`之后，`test_suite`元数据就确定不变了，所以它也非常适合写入`paster`的模板中去，在构建目录的时候自动生成。

建议75： 利用测试驱动开发提高代码的可测性

测试驱动开发（**Test Driven Development**，**TDD**）是敏捷开发中一个非常重要的理念，它提倡在真正开始编码之前测试先行，先编写测试代码，再在其基础上通过基本迭代完成编码，并不断完善。其目的是编写可用的干净的代码。所谓可用就是能够通过测试满足基本功能需求，而干净则要求代码设计良好、可读性强、没有冗余。在软件开发的过程中引入**TDD**能带来一系列好处，如改进的设计、可测性的增强、更松的耦合度、更强的质量信心、更低的缺陷修复代价等。那么，如何在编程过程中实施测试驱动开发呢？一般来说，遵循如图7-2所示的过程。

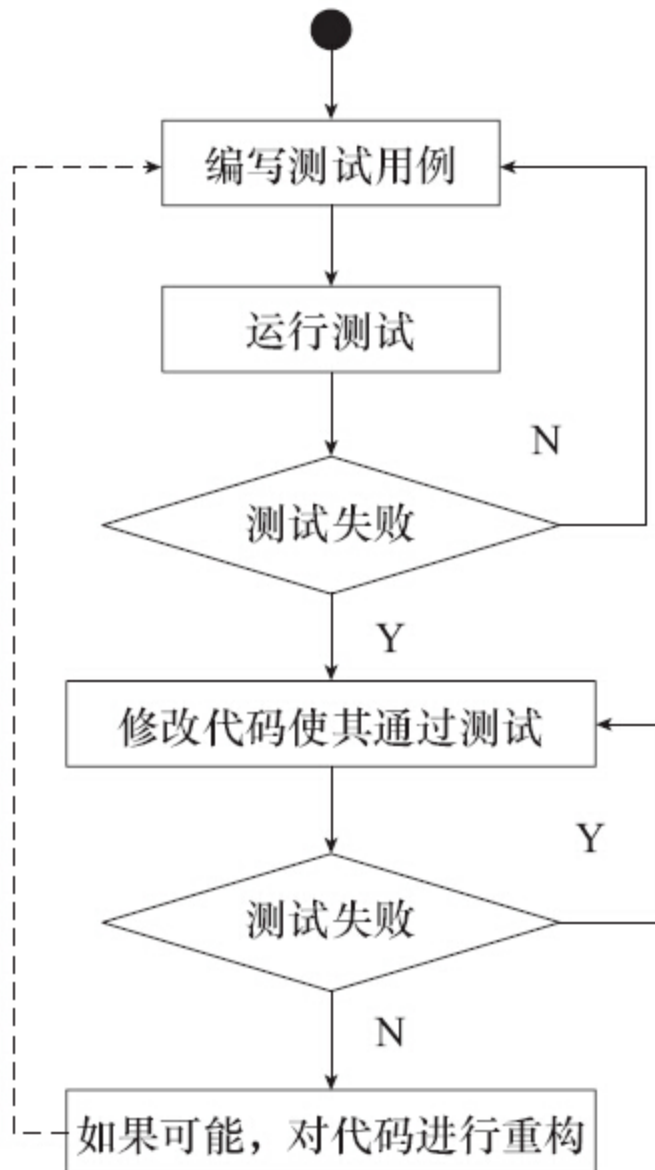


图7-2 测试驱动开发流程

- 1) 编写部分测试用例，并运行测试。
- 2) 如果测试通过，则回到测试用例编写的步骤，继续添加新的测试用例。
- 3) 如果测试失败，则修改代码直到通过测试。

4) 当所有测试用例编写完成并通过测试之后，再来考虑对代码进行重构。

假设开发人员需要编写一个求输入数列的平均数的例子。在开始编写测试用例之前我们先编写简单的编码以保证测试能真正开始执行。

假设源文件avg.py内容如下:

```
def avg(x):  
    pass
```

当完成基本的代码框架之后，便可以开始实施TDD的具体过程了。

步骤1 编写测试用例。基本的测试用例应该包括对整数、浮点数、混合输入情况下基本功能的验证，以及对空输入、无效输入的处理。测试用例代码如下:

```
import unittest  
from avg import avg  
class TestAvg(unittest.TestCase):  
    def test_int(self):  
        print "test average of integers:"  
        self.assertEqual(avg([0,1,2]),1)  
    def test_float(self):  
        print "test average of float:"  
        self.assertEqual(avg([1.2,2.5,0.8]),1.5)  
    def test_empty(self):  
        print "test empty input:"  
        self.assertFalse(avg([]),False)  
    def test_mix(self):  
        print "test with mix input:"  
        self.assertEqual(avg([-1,3,7]),3)  
    def test_invalid(self):  
        print "test with invalid input:"  
        self.assertRaises(TypeError,avg,[-1,3,[1,2,3]])  
if __name__ == '__main__':  
    unittest.main()
```

步骤2 运行测试用例（部分输出），测试结果显示失败，如图7-3所示。

```
test_empty <testavg.TestAvg> ... test empty input:
ok
test_float <testavg.TestAvg> ... test average of float:
FAIL
test_int <testavg.TestAvg> ... test average of integers:
FAIL
test_invalid <testavg.TestAvg> ... test with invalid input:
FAIL
test_mix <testavg.TestAvg> ... test with mix input:
FAIL
```

图7-3 测试结果

步骤3 开始编码直到所有的测试用例都通过。这是一个不停地重复迭代的过程，需要多次重复编码测试，直到上面的测试用例全部执行成功。

```
def avg(x):
    if len(x)<=0:
        print "you need input at least one number"
        return False
    sum = 0
    try:
        for i in x:
            sum += i
    except TypeError:
        raise TypeError("your input is not value with unsupported type")
    return sum/len(x)
```

步骤4 重构。在测试用例通过测试之后，现在可以考虑一下重构的问题了。代码有没有更优化的实现方式呢？显然直接利用内建函数sum更高效直接。因此对代码进行重构并重新测试生成最终产品代码。

```
def avg(*x):
    if len(*x)<=0:
        print "you need input at least one number"
        return False
```

```
try:
    return sum(*x)/len(*x)
except TypeError:
    raise TypeError("your input is not value with unsupported type")
```

关于测试驱动开发和提高代码可测性方面有以下几点需要说明：

- TDD**只是手段而不是目的，因此在实践中尽量只验证正确的事情，并且每次仅仅验证一件事。当遇到问题时不要局限于**TDD**本身所涉及的一些概念，而应该回头想想采用**TDD**原本的出发点和目的是什么。

- 测试驱动开发本身就是一门学问，不要指望通过一个简单的例子就掌握其精髓。如果需要更深入的了解，推荐在阅读相关书籍的同时在实践中不断提高对其的领悟。

- 代码的不可测性可以从以下几个方面考量：实践**TDD**困难；外部依赖太多；需要写很多模拟代码才能完成测试；职责太多导致功能模糊；内部状态过多且没有办法去操作和维护这些状态；函数没有明显返回或者参数过多；低内聚高耦合；等等。如果在测试过程中遇到这些问题，应该停下来想想有没有更好的设计。

建议76：使用Pylint检查代码风格

如果你的团队遵循PEP8的编码风格，Pylint是个不错的选择（当然还有其他很多选择，如pychecker、pep8等）。Pylint始于2003年，是一个代码分析工具，用于检查Python代码中的错误，查找不符合代码编码规范的代码以及潜在的问题。支持不同的OS平台，如Windows、Linux、OSX等，目前最新的版本为1.0.0。其特性如下：

- 代码风格审查**。它以Guido van Rossum的PEP8为标准，能够检查代码的行长度，不符合规范的变量名以及不恰当的模块导入等不符合编码规范的代码。

- 代码错误检查**。如未被实现的接口，方法缺少对应参数，访问模块中未定义的变量等。

- 发现重复以及设计不合理的代码，帮助重构**。

- 高度的可配置化和可定制化**，通过对pylintrc文件的修改可以定义自己适合的规范。

- 支持各种IDE和编辑器集成**。如Emacs、Eclipse、WingIDE、VIM、Spyder等。读者可以查看<http://docs.pylint.org/ide-integration>获取更为详细的信息。

- 能够基于Python代码生成UML图**。Pylint 0.15中就集成了Pyreverse，能够轻易生成UML图形。感兴趣的读者可以查看<http://www.logilab.org/blogentry/6883>。

·能够与Hudson、Jenkins等持续集成工具相结合支持自动代码审查。

下面我们来看如何使用Pylint进行代码检查。以求平衡数为例（平衡数的定义：在一个列表中该数字之前的所有数之和与该数之后的所有数之和相等）。代码如下：

```
""" This script is used for testing balance point in a list """
def main():
    """
    This program is used to find the balance point,the defenition of the balance
    point is that:
    In a array,the sum of all the number before present point is equal to the sum
    of the number after present point
    """
    numbers = [1,3,5,7,8,25,4,20,29]
    sum = 0
    for num in numbers:
        sum += num
        print ("The present number is",num,"\n")
    for index in range(len(numbers)):
        print ("The present index is",index,"\n")
        former = 0
        after = 0
        i = 0
        for i in range(index):
            former += numbers[i]
        after = sum - former - numbers[index]
        if(former == after):
            print ("The balance point is:", numbers[index])
if __name__ == "__main__":
    main()
```

使用Pylint分析代码，输出分为两部分：一部分为源代码分析结果，第二部分为统计报告。报告部分主要是一些统计信息，总体来说有以下6类：

- 1) **Statistics by type:** 检查的模块、函数、类等数量，以及它们中存在文档注释以及不良命名的比例。
- 2) **Raw metrics:** 代码、注释、文档、空行等占模块代码量的百分比统计。
- 3) **Duplication:** 重复代码的统计百分比。

4) **Messages by category:** 按照消息类别分类统计的信息以及和上一次运行结果的对比。

5) **Messages:** 具体的消息ID及它们出现的次数，如C0303出现29次。

6) **Global evaluation:** 根据公式计算出的分数统计10.0-
 $((\text{float}(5 * \text{error} + \text{warning} + \text{refactor} + \text{convention}) / \text{statement}) * 10)$ 。

我们来重点讨论一下源代码分析主要以消息的形式显示代码中存在的问题。消息以MESSAGE_TYPE:LINE_NUM:[OBJECT:]MESSAGE的形式输出，如图7-4所示。主要分为以下5类：

- (C)惯例。违反了编码风格标准。
- (R)重构。写得非常糟糕的代码。
- (W)警告。某些Python特定的问题。
- (E)错误。很可能是代码中的bug。
- (F)致命错误。阻止Pylint进一步运行的错误。

```
***** Module BalancePoint
1, 0: Trailing whitespace <trailing-whitespace>
2, 0: Trailing whitespace <trailing-whitespace>
3, 0: Trailing whitespace <trailing-whitespace>
4, 0: Found indentation with tabs instead of spaces <mixed-indentation>
4, 0: Trailing whitespace <trailing-whitespace>
```

图7-4 Pylint输出信息

比如图7-4第一行信息输出trailing-whitespace信息，如果想进一步弄清楚这个信息所表达的意思，可以使用命令pylint --help-msg="trailing-whitespace"来查看。

```
No config file found, using default configuration
:C0303 (trailing-whitespace): *Trailing whitespace*
    Used when there is whitespace between the end of a line and the newline. This
    message belongs to the format checker.
```

这里提示的是行尾存在空格，也许对很多人来说这类格式检查过于严格。再比如上面的检查中输出较多的是W0312（使用Tab键而不是空格键作为缩进）相关的错误，如果不希望对这类代码风格进行检查，可以使用命令行过滤掉这些类别的信息，如`pylint -d C0303,W0312 BalancePoint.py`，输出结果如图7-5所示。

```
***** Module BalancePoint
C:  5, 0: Line too long (91/80) (line-too-long)
C:  6, 0: Line too long (111/80) (line-too-long)
W:  9, 1: Redefining built-in 'sum' (redefined-builtin)
C:  8, 1: Comma not followed by a space
      numbers = [1,3,5,7,8,25,4,20,29]
              ^^ (no-space-after-comma)
C: 12, 2: Comma not followed by a space
      print ("The present number is",num,"\n")
              ^^ (no-space-after-comma)
C: 15, 2: Comma not followed by a space
      print ("The present index is",index,"\n")
              ^^ (no-space-after-comma)
```

图7-5 过滤掉某些类别信息后的输出

根据信息提示做如下修改再运行便不再有其他问题，而Global evaluation的评估为10。

- 1) 调整第五、第六行代码长度。
- 2) `sum`名称和内建函数`sum`同名，修改为`result`。
- 3) 在列表后面增加空格，`numbers = [1, 3, 5, 7, 8, 25, 4, 20, 29]`。

4) 在12行输出语句后面加上空格`print ("The present index is", index, "\n")`。

5) 在12行输出语句后面加上空格`print ("The present number is", num, "\n")`。

PyLint支持可配置化，如果在项目中希望使用统一的代码规范而不是默认的风格来进行代码检查，可以指定`--generate-rcfile`来生成配置文件。默认的**Pylintrc**可以在**PyLint**的目录**examples**中找到。如默认支持的变量名的正则表达式为：`variable-rgx=[a-z_][a-z0-9_]{2,30}$`。如果希望改为只能以2~10个字符的小写字母命名，可以将代码修改为`variable-rgx=[a-z_]{2,10}$`。其他配置如**reports**用于控制是否输出统计报告；**max-module-lines**用于设置模块最大代码行数；**max-line-length**用于设置代码行最大长度；**max-args**用于设置函数的参数个数等。读者可以自行查看**pylintrc**文件，获取更为详细的信息。

建议77：进行高效的代码审查

“代码审查是件很无聊的事情，费时费力，纯粹形式主义”，有这样想法的开发人员不在少数，甚至很多团队由于这些原因或者开发进度等其他因素干脆就直接忽略这个环节。那么，是不是代码审查真的那么一无是处呢？真相是：它远比你想象的要重要得多。很多人之所以会产生这样的误区，多半是因为代码审查的流程不够高效或者审查的目的出现了偏差。我们通过一组统计数据来看严格高效的代码审查到底有多重要，如图7-6所示。

	假设存在10000个bug		修复一个bug的代价(S)	总体代价	
代码审查	10%	60%	75	75 000	450 000
	9000	4000			
单元测试	20%	40%	150	345 000	690 000
	7200	2400			
功能测试	60%		500	2505 000	1410 000
	2880	960			
系统测试	70%		2000	6537 000	2754 000
	864	288			
客户汇报	30%		25 000	10 877 000	4 914 000
	604	201			

图7-6 不同代码审查效率下修复bug的代价

图7-6所表示的是在不同的阶段中未经过严格有效的代码审查和经过高效代码审查后所得到的bug发现比率和修复代价对比关系。从图7-6可以看出，有效的代码审查流程非常必要，它能够使得大量bug在该阶段被发现，并且大幅度降低bug修复的总体代价，因为代码审查阶段bug的修复代价非常小。除此之外，代码审查还有助于建立高效的团队，提高团队成员之间的协作能力以及编码水平，有助于跨团队之间知识共享。那么，团队应该以什么样的态度去看待代码审查呢？

1) 不要错误地理解代码审查会的目的。代码审查会的首要目的是提高代码质量，找出defect或者设计上的不足而非修复defect。也许有人会疑虑，defect不要修复吗？要！但绝不是审查会上，这是保证代码审查高效的第一个关键，因为时间有限，不可能也不允许在审查会上去讨论bug或者defect如何修复，所有类似问题应该记录下来等会后讨论。

2) 代码审查过程不应该有KPI（Key Performance Indicator）评价的成分。如果将代码审查中发现bug的概率作为对开发人员绩效考核的标准，势必会打击参与人员的积极性，有些人可能因为怕承担责任而不愿意直接指出代码中的问题。同时也会引起被审查人员的自我保护意识，当问题发生的时候，被审查者可能更关注于怎么推卸问题而不是静下心来解释问题产生的原因，以及如何改进。因此，代码审查要努力做到针对技术和问题本身。

3) 对管理层的建议：除非经理或者组长真正参与到具体的技术问题，否则应该尽量避免这些人员参与代码审查会，因为这些人直接与员工KPI评价挂钩，即使申明不会用bug或者defect发现的数目来作为评价标准，但由于这些人身份的特殊，仍然可能造成评审参与人员不能诚实面对代码本身的局面。

4) **对开发者的建议：** 把代码审查当做一个学习的机会，而不要看成是浪费你的时间帮别人来解决问题。无论你技术水平的高低，阅读和审查同行的代码可以让你学习更优秀的设计和编码，也能让你更快速地知道如何避免一些糟糕的代码和设计上的缺陷。

有了正确的态度，还需要一定的流程来保证才能做到高效的代码审查。

(1) 定位角色。

一般来说代码审查会上有4类角色：仲裁者、会议记录者、被评审开发人员和评审者。

1) **仲裁者：** 也叫主持人，一般由技术专家或者资深技术人员担任。担任该角色的人员不仅要技术精湛、业务精通，而且要有全局系统思维和较好的沟通以及领导能力。他的主要职责是控制会议流程和时间，保证会议流程的高效性，同时能够在必要的时候给予评审技术指导。因此在评审过程中如果出现了参与人员就一个问题争论不休的时候，该角色应该能够给予及时制止和指导性意见，特别要注意保护被评审人员。在评审结束之后，仲裁者还应该确保发现的问题被正确解决。

2) **会议记录者：** 及时记录评审过程中发现的问题，包括问题的提出者、问题描述、问题产生的位置，如果有解决思路还应该记录解决方法，如果问题暂时没有解决办法应该记录下一步行为，如是会后研究还是另外开会讨论等。需要强调的是，在记录完一个问题相关信息之后，会议记录者必须和被评审开发人员确认记录内容，以保证记录的信息准确无误且被该开发人员认可。

3) 被评审开发人员：一般被评审开发人员在评审开始的时候应该对其代码有综合性的介绍，在评审者提出问题或者质疑的时候应该及时给出解释。被评审开发人员对其他人提出的意见或者问题要正确看待，不要当做攻击，记住：所有人的目的都是为了开发出质量更高的软件。被评审开发人员一般不参与对代码的具体审查。

4) 评审者：除了以上3类人员外，其余与会的人都称为评审者，评审者组成应该是有经验的和经验相对较弱的人员兼而有之，因为这样的组合才能在评审的同时更好地起到培训作用，提高经验较弱人员的编码能力。

(2) 充分准备

在代码提交审查之前，代码作者需要做以下准备：对代码进行自我修正，包括对代码风格进行检查、添加必要的注释、完成必要的功能测试等；提前通知参与审查的人员，以便他们能够事先对代码框架有个基本了解；确定会议时间和进程。

执行语句	i	j	sum
	0		0
j=input("num:")		2	
i<j is True			
sum=i+j			3
j=input("num:")		-1	
i<j is Fasle			
sum=j			-1

图7-7 台面检查示例

(3) 合理使用技术和工具

代码审查并不是完全依靠经验的，有一些方法可以遵循。常见的方法和工具如下。

①检查表（**checklist**）：检查表有利于有针对性地发现代码中存在的问题，如变量是否初始化，函数调用的参数，命名是否一致，字符串是否正确解码，有没有**import**未使用的**lib**，逻辑操作符是否正确，**()**、**{}**对是否一致等。

②台面检查（**Desk Checking**）：适合在编码早期对顺序执行的代码进行检查，手工模拟代码的执行过程来检查程序中潜在的问题。图7-7所示就是下面的代码的台面检查。

```
i = 0
j = input('num:')
sum = 0
if i < j:
    sum = i+j
else:
    sum = j
```

③IRT（**Interleaving Review Technique**）：适合于并发性的代码或者容错性系统。更多IRT的资料读者可以参考https://www.research.ibm.com/haifa/Workshops/PADTAD2004/papers/irt_padtad2.pdf。

④代码审查工具：如Rietveld、review board、Collaborative Code Review Tool（CCRT）等。

(4) 控制评审时间和评审内容

为了保证效率，一般来说一次评审时间要尽量控制在45分钟到1小时。研究表明，当人的注意力集中超过1小时，效率就会急剧下降。而一次评审的代码行数应控制在200行以内，最好不超过400行。如图7-8所示描述的就是缺陷发现的密度与评审代码行数之间的关系。由图可知当被评审的代码超过200行时，查出缺陷的密度就会急剧下降。

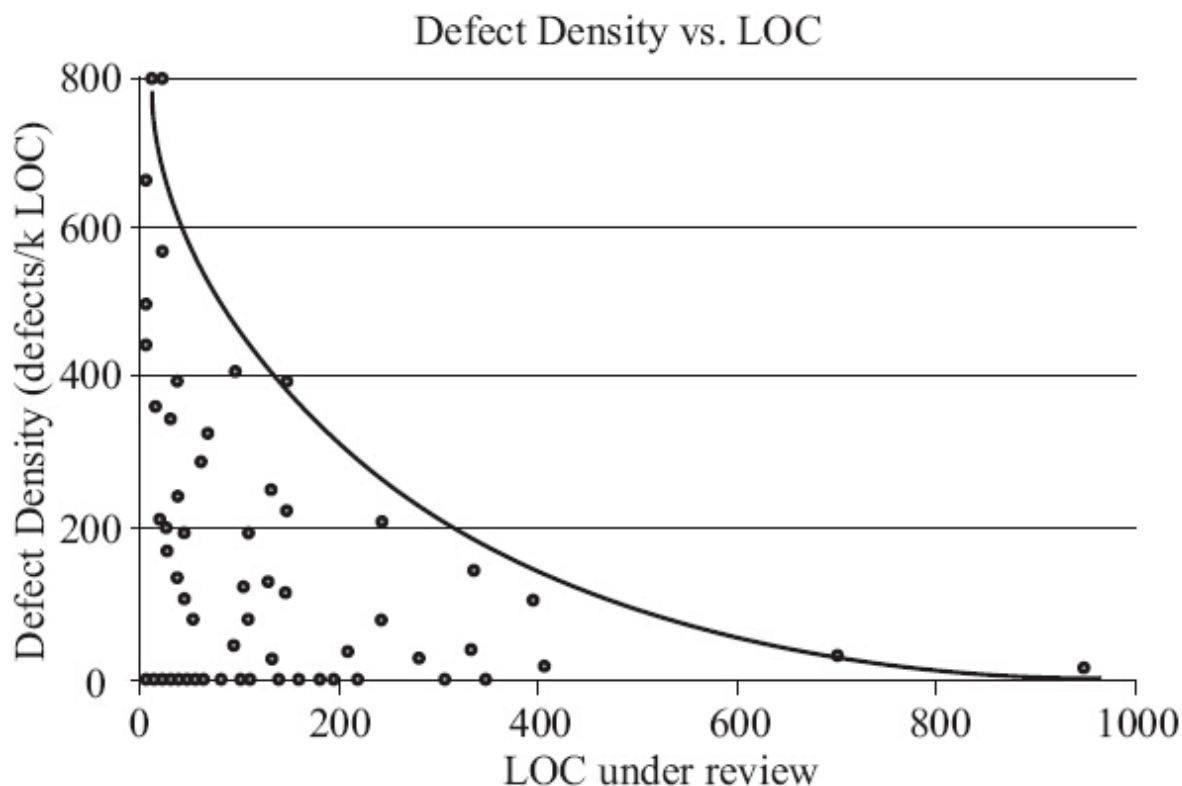


图7-8 代码行数与查出缺陷密度之间的关系

(5) 关注技术层面，对事不对人

要把重点放在技术问题以及如何解决上，而不是诸如代码风格（代码风格当然重要，但应该在评审之前就由开发者对照组织规定事先完成，评审过程附带指出一些特殊问题）、时间进度之类的非技术层面的问题上。此外，评审人员应该对事不对人，在评审过程中要合

理使用一些客观的语言，如“我没读懂这段代码的意思”，或者“我认为怎样做更好”、而不是“你这代码写的太烂了”、“垃圾”等带有攻击性的语言。较好的方法是评审者在评审的过程中时刻警惕：代码是不是正常工作？与需求是不是匹配？实现上有没有潜在的问题和风险？推荐采取“六顶帽子”思考法。

（6）记录问题，追踪进一步行动

记录问题是为了保证在评审会上发现的问题和缺陷在会后能得到及时的修复。因此会议记录者应该在会后及时将会议记录发送给相关人员，并保证后续行动都及时实施。

（7）不要忽视附加的培训作用

评审是手段，发现代码缺陷，提高代码质量和团队人员的编码水平才是目的，因此评审过程中别忘了培训的附加作用，仲裁者应该针对优秀的代码鼓励其他参与者借鉴，而对于一些警示问题代码要提醒参与者避免。

建议78：将包发布到PyPI

建立项目之后，添加了相应的业务代码，并通过测试之后，就可以考虑发布给下游用户了。如果是项目内部协作，把项目打一个zip包或者tar ball发出去，最简单不过了。不过尽管如此简单，setuptools仍然提供了完善的支持。

```
>sudo python setup.py sdist --formats=zip,gztar
running sdist
running check
creating arithmetic-1.0
making hard links in arithmetic-1.0...
hard linking arithmetic.py -> arithmetic-1.0
hard linking setup.py -> arithmetic-1.0
creating dist
creating 'dist/arithmetic-1.0.zip' and adding 'arithmetic-1.0' to it
adding 'arithmetic-1.0/arithmetic.py'
adding 'arithmetic-1.0/PKG-INFO'
adding 'arithmetic-1.0/setup.py'
Creating tar archive
removing 'arithmetic-1.0' (and everything under it)
```

setuptools的sdist命令的意思是构建一个源代码发行包，它将根据调用setup()函数时给定的实参将整个项目打包（和压缩）。根据当前的平台（操作系统）不同，产出的文件也是不一样的。一般在MS Windows系统下，产生.zip格式的压缩包，而在GNU Linux或Mac OS X系统下，产生.tar.gz格式的压缩包。考虑到最终安装程序包的用户可能在不同的系统下使用，需要产品指定（或更多）格式的包文件，可以使用--formats参数。如上列指定产生.zip格式和.tar.gz格式。最终产生的包文件放在./dist目录下。

```
>ls dist
arithmetic-1.0.tar.gz arithmetic-1.0.zip
```

产生这两个包以后，就可以发布给项目的下游合作者了。发布方式可以是邮件、FTP，或者直接使用IM传送。下游开发者收到后有两种安装方式：一种是解压缩，然后进入setup.py文件所在的目录执行python setup.py install命令安装；另一种是使用pip安装，执行pip install arithmetic-1.0.tar.gz即可。

对于个人项目或者迷你团队而言，通过邮件、FTP或者IM发布无可厚非，但如果是较大的团队一起协作一个项目，那么最好是把包发布到PyPI上面。可以是pypi.python.org这个官方的PyPI，也可以是团队架设的私有PyPI。在这里，先讲一下怎么把包发布到官方的PyPI。

其实标准库distutils自身已经带有发布到PyPI的功能，那就是register和upload命令。

```
$ python setup.py --help-commands
Standard commands:
...
  register          register the distribution with the Python package index
...
  upload            upload binary package to PyPI
...
```

register命令用以在PyPI上面注册一个包，这个包名必须是尚未使用过的。在注册包名之前，先在PyPI上注册一个用户，可以通过PyPI网页注册，也可以直接使用register命令提供的选项注册。

```
$ python setup.py register
...
We need to know who you are, so please choose either:
 1. use your existing login,
 2. register as a new user,
 3. have the server generate a new password for you (and email it to you), or
 4. quit
Your selection [default 1]:
```

上面第2个选项就是用来注册新用户的，选中之后向导将会指引用户输入用户名、密码和邮箱等信息，很快就可以注册完成，在此不展

开说了。如果已经有了PyPI账号，那么选择第1个选项，输入用户名和密码，验证通过以后，distutils向PyPI申请注册包名，一般都能够成功。但如果这个包名已经被别的用户使用过了，那会引发一个403错误，指出你不能把这个包的信息存储到PyPI。

```
$ python setup.py register -n
running register
running check
...
Registering arithmetic to http://pypi.python.org/pypi
Server response (200): OK
```

包名注册之后，就可以把包上传到PyPI了。

```
$ python setup.py sdist upload
running sdist
running check
writing manifest file 'MANIFEST'
creating arithmetic-1.0
making hard links in arithmetic-1.0...
hard linking arithmetic.py -> arithmetic-1.0
hard linking setup.py -> arithmetic-1.0
Creating tar archive
removing 'arithmetic-1.0' (and everything under it)
running upload
Submitting dist/arithmetic-1.0.tar.gz to http://pypi.python.org/pypi
Server response (200): OK
```

上传之后，就可以通知合作者使用setuptools/pip安装了。

```
pip install arithmetic
```

第8章 性能剖析与优化

“选择了脚本语言就要忍受其速度”，这句话在某种程度上说明了Python作为脚本语言在效率和性能方面的一个不足之处。但这并没有你想象的那么夸张，也不是说你只能坐以待毙。性能与效率与很多方面息息相关，如软件设计、运行环境、网络带宽、更优化的代码等。代码优化并不神秘，它建立在软件算法和硬件体系结构等知识层面之上，有一定的方法可以遵循。本章将着重从代码层面来探讨如何提高Python的效率和性能，并重点分析一些常见的优化方法和技巧。

建议79：了解代码优化的基本原则

代码优化是指在不改变程序运行结果的前提下使得程序运行的效率更高，优化的代码意味着运行速度更快或者占有的资源更少。进行代码优化时需要记住以下几点原则。

(1) 优先保证代码是可工作的

Donald Knuth曾说过，过早优化是编程中一切“罪恶”的根源。很多人热衷优化，一开始写代码就奔着性能这个目标。但事实真相是“让正确的程序更快要比让快速的程序正确容易得多。”因此优化的前提是代码满足了基本的功能需求，是可工作的。过早地进行优化可能会忽视对总体性能指标的把握，忽略可移植性、可读性、内聚性等，更何况每个模块甚至每行优化的代码并不一定能够带来整体运行性能良好，因为性能瓶颈可能出现在意想不到的地方，如模块与模块之间的交互和通信等，在得到整体视图之前不要主次颠倒。需要说明的是，这并不是不鼓励你在代码实现的过程中去尝试更优的实现方式，在编码的过程中同样应该遵循Python的哲学和本书前面章节所提倡的风格与语法，尽量选择更好的算法或者实现。

(2) 权衡优化的代价

优化是有代价的，想解决所有性能问题几乎是不可能的。从代码本身的角度来讲，可能面临着牺牲时间换空间或者牺牲空间换时间的抉择；从项目的角度来讲，质量、时间和成本这三者之间“铁三角”关系不会改变，如果性能是权衡质量的一个指标的话，更好的性能意味

着需要更多的时间和人力或者更强大的硬件资源；从用户的角度来看，根据80/20法则，最终影响用户体验的可能也就是20%的性能问题。因此，优化需要权衡代价，如果在项目时间紧迫的情况下能够仅仅通过增加硬件资源就解决主要性能问题，不妨选择更强大的部署环境；或者在已经实现的代码上进行修修补补试图进行优化代码所耗费的精力超过重构的代价时，重构可能是更好的选择。

(3) 定义性能指标，集中力量解决首要问题

你可能曾经听到过客户这样的声音：我希望这个功能反应更快一点。这很好，起码你明白优化的目标所在，而不至于像个“无头苍蝇”一样抓不住客户需要的方向而导致最后落个费力不讨好的结果。但这还不够好，为什么？什么标准才符合更快一点这个说法呢？更快到底是多快？“一千个人心中就有一千个哈姆雷特”，我们必须制定出可以衡量快的具体指标，比如在什么样的运行环境下（如网络速度、硬件资源等）、运行什么样的业务响应时间的范围是多少秒。这里要着重强调的是：精确，可度量。更快、非常快这些都是描述性的词语，并不可度量，不同的人有着不同的衡量标准，可能对于一个请求你认为2秒内能够返回结果已经够快了，但业务人员所理解的够快可能是0.5秒内，偏差由此产生。如果你的客户并不能提出专业精确的目标，那么相关需求人员或者技术人员也一定要引导和帮助客户（如运用SMART法则等）最终达成契约。另外，性能优化一定要站在客户和产品本身的角度上分析而不是开发人员的角度上。为什么？因为客户才是我们服务的主要对象，他们的想法才能代表最终的需求。比如开发人员可能觉得安装的时间过长而花费不少精力进行优化，但客户真正关心的可能是在系统上部署一个新的服务的响应时间。因此，在进行优化之前，一定要针对客户关心的问题主次排列，并集中力量解决主要问题。

(4) 不要忽略可读性

优化不能以牺牲代码的可读性，甚至带来更多的副作用为代价。实际应用中经常运行的代码可能只占一小部分，但几乎所有代码都是需要维护的，因此在代码的可读性、可维护性以及更优化的性能之间需要权衡。如果优化的结果是使代码变得难以阅读和理解，可能停止优化或者选择其他替代设计更好。

最后需要说明的是，优化无极限，不要陷入怪圈，什么时候应该优化、什么时候应该停止优化心里得有谱，性能较优的代码确实很吸引人，但过犹不及。

建议80：借助性能优化工具

“工欲善其事，必先利其器”，好的工具能够对性能的提升起到非常关键的作用。常见的性能优化工具有Psyco、Pypy和cPython等。本节我们将简单讨论前两者，cPython性能优化具体使用见建议90一节。

(1) Psyco

Psyco是一个just-in-time的编译器，它能够在不改变源代码的情况下提高一定的性能，Psyco将操作编译成部分优化的机器码，其操作分成三个不同的级别，有“运行时”、“编译时”和“虚拟时”变量，并根据需要提高和降低变量的级别。运行时变量只是常规Python解释器处理的原始字节码和对象结构。一旦Psyco将操作编译成机器码，那么编译时变量就会在机器寄存器和可直接访问的内存位置中表示。同时Python能高速缓存已编译的机器码以备以后重用，这样能节省一点时间。但Psyco也有其缺点，如，其本身运行所占内存较大。2012年3月12日，Psyco项目主页上宣布Psyco停止维护并正式结束，由Pypy所接替。到结束为止，Psyco也没有提供对Python2.7版本的支持。对Psyco感兴趣的读者可以参考其主页（<http://psyco.sourceforge.net/>）了解更多信息。

(2) Pypy

Python的动态编译器，是Psyco的后继项目。其目的是，做到Psyco没有做到的动态编译。Pypy的实现分为两部分：第一部分“用Python实现的Python”，这里虽然是这么说，但实际上它是使用一个名

为RPython的Python子集实现的，PyPy能够将Python代码转成C、.NET、Java等语言和平台的代码；第二部分PyPy集成了一种编译rPython的即时（JIT）编译器，和许多编译器、解释器不同，这种编译器不关心Python代码的词法分析和语法树，因为它用Python语言写的，所以它直接利用Python语言的Code Object（Python字节码的表示），也就是说，PyPy直接分析Python代码所对应的字节码，这些字节码既不是以字符形式也不是以某种二进制格式保存在文件中。如图8-1所示是针对同一段代码分别使用Python和PyPy运行得到的时间消耗示意图。

从图8-1中可见看出，使用PyPy来编译和运行程序，随着运算规模的扩大，其效率显著提高。

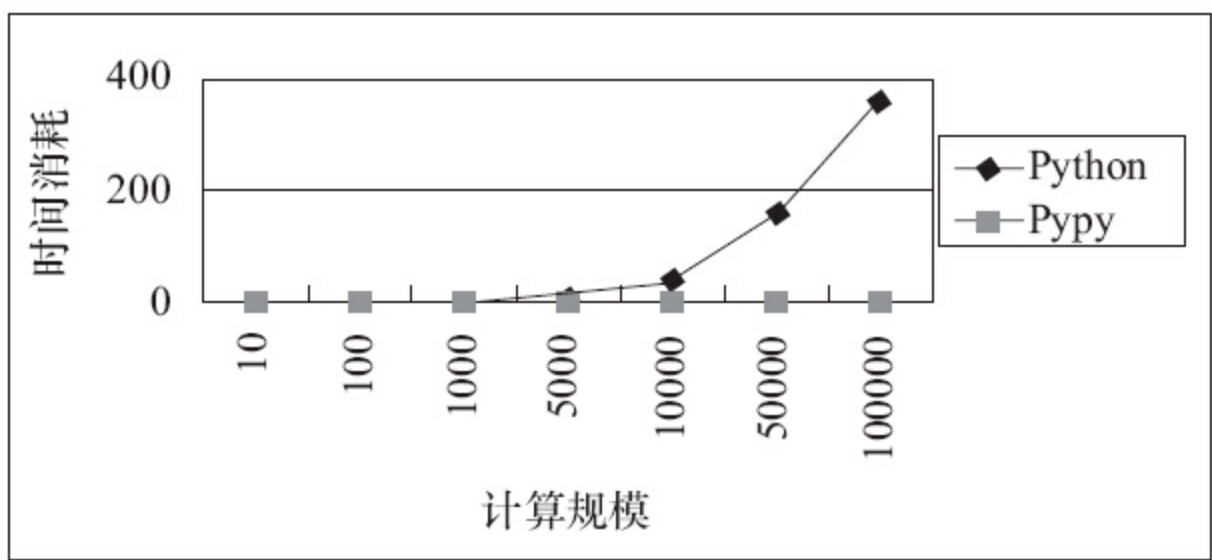


图8-1 Python与PyPy运行时间比较

建议81： 利用cProfile定位性能瓶颈

程序运行慢的原因有很多，但真正的原因往往是一两段设计并不那么良好的不起眼的程序，比如对一系列元素进行自定义的类型转换等。程序性能影响往往符合80/20法则，即20%的代码的运行时间占用了80%的总运行时间（实际上，比例要夸张得多，通常是几十行代码占用了95%以上的运行时间），所以如何定位瓶颈所在很有难度，靠经验是很难找出造成性能瓶颈的代码的。这时候，我们需要一个工具帮忙，下文通过cProfile分析相关的独立模块，基本上解决了定位性能瓶颈问题。

profile是Python的标准库。可以统计程序里每一个函数的运行时间，并且提供了多样化的报表，而cProfile则是它的C实现版本，剖析过程本身需要消耗的资源更少。所以在Python 3中，cProfile代替了profile，成为默认的性能剖析模块。使用cProfile来分析一个程序很简单，以下面一个程序为例：

```
sum = 0
for i in range(100):
    sum += i
return sum
if __name__ == "__main__":
    foo()
```

现在要用profile分析这个程序。很简单，把if程序块改为如下：

```
if __name__ == "__main__":
    import cProfile
    cProfile.run("foo()")
```

我们仅仅是import了cProfile这个模块，然后以程序的入口函数名为参数调用了cProfile.run这个函数。程序运行的输出如下：

5 function calls in 0.143 CPU seconds					Ordered by: standard name
ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	:0(range)
1	0.143	0.143	0.143	0.143	:0(setprofile)
1	0.000	0.000	0.000	0.000	<string>:1(?)
1	0.000	0.000	0.000	0.000	prof1.py:1(foo)
1	0.000	0.000	0.143	0.143	profile:0(foo())
0	0.000		0.000		profile:0(profiler)

上面显示了prof1.py里函数调用的情况，根据数据我们可以清楚地看到foo()函数占用了100%的运行时间，foo()函数是这个程序里名副其实的热点。

除了用这种方式，cProfile还可以直接用Python解释器调用cProfile模块来剖析Python程序。如在命令行界面输入如下命令：

```
python -m cProfile prof1.py
```

产生的输出跟直接修改脚本调用cProfile.run()函数有一样的功效。

cProfile的统计结果分为ncalls、tottime、percall、cumtime、percall、filename:lineno(function)等若干列，如表8-1所示。

表8-1 cProfile 的统计结果以及各项意义

统 计 项	意 义
ncalls	函数的被调用次数
tottime	函数总计运行时间，不含调用的函数运行时间
percall	函数运行一次的平均时间，等于 tottime/ncalls
cumtime	函数总计运行时间，含调用的函数运行时间
percall	函数运行一次的平均时间，等于 cumtime/ncalls
filename:lineno(function)	函数所在的文件名、函数的行号、函数名

通常情况下，cProfile的输出都直接输出到命令行，而且默认是按照文件名排序输出的。这就给我们造成了障碍，我们有时候希望能够把输出保存到文件，并且能够以各种形式来查看结果。cProfile简单地

支持了一些需求，我们可以在cProfile.run()函数里再提供一个实参，就是保存输出的文件名。同样，在命令行参数里，我们也可以加多一个参数，用来保存cProfile的输出。

cProfile解决了我们的对程序执行性能剖析的需求，但还有一个需求：以多种形式查看报表以便快速定位瓶颈。我们可以通过pstats模块的另一个类Stats来解决。Stats的构造函数接受一个参数——就是cProfile的输出文件名。Stats提供了对cProfile输出结果进行排序、输出控制等功能。如我们把前文的程序改为如下：

```
#
...略
if __name__ == "__main__":
    import cProfile
    cProfile.run("foo()", "prof.txt")
    import pstats
    p = pstats.Stats("prof.txt")
    p.sort_stats("time").print_stats()
```

引入pstats之后，将cProfile的输出按函数占用的时间排序，输出如下：

```
Sun Jan 14 00:03:12 2007      prof.txt
      5 function calls in 0.002 CPU seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
      1   0.002    0.002    0.002    0.002   :0(setprofile)
      1   0.000    0.000    0.002    0.002  profile:0(foo())
      1   0.000    0.000    0.000    0.000  G:/prof1.py:1(foo)
      1   0.000    0.000    0.000    0.000  <string>:1(?)
      1   0.000    0.000    0.000    0.000  :0(range)
      0   0.000          0.000          profile:0(profiler)
```

Stats有若干个函数，这些函数组合能输出不同的cProfile报表，功能非常强大，如表8-2所示。下面简单地介绍一下这些函数。

表8-2 Stats函数以及对应作用

函 数	函数的作用
strip_dirs()	用以除去文件名前名的路径信息
add(filename,[...])	把 profile 的输出文件加入 Stats 实例中统计
dump_stats(filename)	把 Stats 的统计结果保存到文件
sort_stats(key,[...])	最重要的一个函数，用以排序 profile 的输出
reverse_order()	把 Stats 实例里的数据反序重排
print_stats([restriction,...])	把 Stats 报表输出到 stdout
print_callers([restriction,...])	输出调用了指定的函数的相关信息
print_callees([restriction,...])	输出指定的函数调用过的函数的相关信息

这里最重要的函数就是`sort_stats`和`print_stats`，通过这两个函数我们几乎可以用适当的形式浏览所有的信息了。下面来详细介绍一下。

1) `sort_stats()`接收一个或者多个字符串参数，如`time`、`name`等，表明要根据哪一列来排序。这相当有用，例如我们可以通过用`time`为`key`来排序得知最消耗时间的函数；也可以通过`cumtime`来排序，获知总消耗时间最多的函数。这样我们优化的时候就有了针对性，可以做到事半功倍了。

`sort_stats`可接受的参数如表8-3所示。

表8-3 `sort_stats`可接受参数列表

参 数	参数对应的意义
ncalls	被调用次数
cumulative	函数运行的总时间
file	文件名
module	模块名
pcalls	简单调用统计（兼容旧版，未统计递归调用）
line	行号
name	函数名
nfl	Name、file、line
stdname	标准函数名
time	函数内部运行时间（不计调用子函数的时间）

2) `print_stats`输出最后一次调用`sort_stats`之后得到的报表。
`print_stats`有多个可选参数，用以筛选输出的数据。`print_stats`的参数可以是数字也可以是Perl风格的正则表达式。相关的内容通过其他渠道了解，这里就不详述啦。仅举以下3个例子：

```
print_stats(".1", "foo:")
```

这个语句表示将`stats`里的内容取前面的10%，然后再将包含“foo:”这个字符串的结果输出。

```
print_stats("foo:", ".1")
```

这个语句表示将`stats`里的包含“foo:”字符串的内容的前10%输出。

```
print_stats(10)
```

这个语句表示将`stats`里前10条数据输出。

实际上，`profile`输出结果的时候相当于如下调用了`Stats`的函数：

```
p.strip_dirs().sort_stats(-1).print_stats()
```

其中`sort_stats`函数的参数是-1，这也是为了与旧版本兼容而保留的。`sort_stats`可以接受-1、0、1、2之一，这4个数分别对应“stdname”、“calls”、“time”和“cumulative”。但如果你使用了数字为参数，那么`pstats`只按照第一个参数进行排序，其他参数将被忽略。

除了编程接口外，`pstats`还提供了友好的命令行交互环境，在命令行执行`python -m pstats`就可以进入交互环境，在交互环境里可以使用`read`或`add`指令读入或加载剖分结果文件，`stats`指令用以查看报表，

callees和callers指令用以查看特定函数的被调用者和调用者。如图8-2所示是pstats的截图，标识了它的基本使用方法。

```
C:\WINNT\system32\cmd.exe - python -m pstats
C:\Documents and Settings\Administrator>python -m pstats
Welcome to the profile statistics browser.
% (help) 在 % 提示符后输入 help 指令查看帮助

Documented commands <type help <topic>>:
=====
EOF add callees callers quit read reverse sort stats strip

Undocumented command 所有可用的指令，可以使用 help XX 查看 XX 指令的帮助
=====
help

% read H:\laiscode\astar\astar.prof 读入部分结果文件
H:\laiscode\astar\astar.prof% stats 0.1 打印报表，只输出 10% 的函数
Wed Jan 30 16:49:12 2008 H:\laiscode\astar\astar.prof

99693 function calls in 1.893 CPU seconds 报表内容

Random listing order was used
List reduced from 38 to 4 due to restriction <0.10000000000000001>

ncalls tottime percall cumtime percall filename:lineno(function)
5713 0.723 0.000 0.723 0.000 H:\laiscode\astar\astar.py:156<code_in_close>
33818 0.166 0.000 0.166 0.000 :0<sqrt>
1 0.000 0.000 1.887 1.887 H:\laiscode\astar\astar.py:2<<module>>
1 0.004 0.004 1.892 1.892 :0<execfile>

H:\laiscode\astar\astar.prof%
```

图8-2 pstats输出信息截图

如果我们某天心血来潮，想知道向list里添加一个元素需要多少时间，或者想知道抛出一个异常需要多少时间，那使用profile就好像用牛刀杀鸡了。这时候一般我们先手动写如下一段代码：

```
import time
def profile():
    bgn = time.time()
    for i in xrange(100000):
        [].append(1)
```

```
        return time.time() - bgn
    print profile()
```

为了测定一条语句，写了好几条代码，真的让人汗颜。更好的选择是**timeit**模块。

timeit除了有非常友好的编程接口，也同样提供了友好的命令行接口。首先来看看编程接口。**timeit**模块包含一个类**Timer**，它的构造函数如下：

```
class Timer( [stmt='pass' [, setup='pass' [, timer=<timer function>]]])
```

stmt参数是字符串形式的一个代码段，这个代码段将被评测运行时间；**setup**参数用以设置**stmt**的运行环境；**timer**可以由用户使用自定义精度的计时函数。

timeit.Timer有3个成员函数，简单介绍如下：

```
timeit( [number=1000000])
```

timeit()执行一次**Timer**构造函数中的**setup**语句之后，就重复执行**number**次**stmt**语句，然后返回总计运行消耗的时间。

```
repeat( [repeat=3 [, number=1000000]])
```

repeat()函数以**number**为参数调用**timeit**函数**repeat**次，并返回总计运行消耗的时间。

```
print_exc( [file=None])
```

`print_exc()`函数用以代替标准的`tracback`，原因在于`print_exc()`会输出错行的源代码。如：

```
>>> t = timeit.Timer("t = foo()/n;print t")
>>> t.timeit()
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in -toplevel-
    t.timeit()
  File "E:/Python27/lib/timeit.py", line 158, in timeit
    return self.inner(it, self.timer)
  File "<timeit-src>", line 6, in inner
    foo()
NameError: global name 'foo' is not defined
```

在这里`NameError`有点让人迷惑，`foo`未定义到底是来自被`timeit`的那段代码还是调用`timeit`的代码本身呢？这个场景就是`print_exc()`函数的用武之地了。

```
>>> try:
    t.timeit()
except:
    t.print_exc()
Traceback (most recent call last):
  File "<pyshell#17>", line 2, in ?
  File "E:/Python27/lib/timeit.py", line 158, in timeit
    return self.inner(it, self.timer)
  File "<timeit-src>", line 6, in inner
    t = foo()
NameError: global name 'foo' is not defined
```

可以看到`traceback`里原来的`foo()`变成了整行代码`t=foo()`，这样丰富的信息能够加速定位错误。

除了可以使用`timeit`的编程接口外，我们也可以在命令行里使用`timeit`，非常方便。

```
python -m timeit [-n N] [-r N] [-s S] [-t] [-c] [-h] [statement ...]
```

其中参数的定义如下：

·`-n N/--number=N`，`statement`语句执行的次数，

·-r N/--repeat=N，重复多少次调用timeit()，默认为3，

·-s S/--setup=S，用以设置statement执行环境的语句，默认为“pass”。

·-t/--time，计时函数，除了Windows平台外默认使用time.time()函数。

·-c/--clock，计时函数，Windows平台默认使用time.clock()函数。

·-v/--verbose，输出更大精度的计时数值。

·-h/--help，简单的使用帮助。

小巧实用的timeit蕴藏了无限的潜能等待你去发掘。如本节开始的例子可以使用一句命令行命令搞定。

```
$ python -m timeit "[]\.append(1)"
1000000 loops, best of 3: 0.187 usec per loop
```

建议82：使用memory_profiler 和 objgraph 剖析内存使用

Python还提供了一些工具可以用来查看内存的使用情况以及追踪内存泄露（如memory_profiler、objgraph、cProfile、PySizer及Heapy等），或者可视化地显示对象之间的引用（如objgraph），从而为发现内存问题提供更直接的证据。本节最后我们再来看看memory_profiler和objgraph这两个工具的使用。

(1) memory_profiler

安装memory_profiler可以使用命令`easy_install-U memory_profiler`或者`pipinstall-U memory_profiler`，也可进行源码安装。需要注意的是，在Windows平台上需要先安装依赖包psutil。memory_profiler的使用非常简单，在需要进行内存分析的代码之前用@profile进行装饰，然后运行命令`python-m memory_profiler 文件名`，便可以输出每一行代码的内存使用以及增长情况。

```
import memory_profiler
@profile
def fibonacci(n):
    .....
```

以代码memory_profiler_test.py为例，输出列分别对应为行号、内存使用情况、内存增长情况以及行所对应的内容。如下所示：

Line #	Mem usage	Increment	Line Contents
=====			
	8.648 MB	0.000 MB	@profile def fibonacci(n):

11.500 MB	2.852 MB	if n < 0:
		return -1
11.500 MB	0.000 MB	elif n <= 1:
11.500 MB	0.000 MB	return 1
11.500 MB	0.000 MB	else:
11.500 MB	0.000 MB	return fibonacci(n -1) + fibo

```
nacci(n -2)
```

更多关于memory_profiler的信息可以参考
https://pypi.python.org/pypi/memory_profiler。

(2) Objgraph

Objgraph的安装非常简单，可以使用命令`pip install objgraph`，或者直接从<https://pypi.python.org/pypi/objgraph>下载进行源码安装。
 Objgraph的功能大致可以分为以下3类：

- 统计。如`objgraph.count(typename[, objects])`表示根据传入的参数显示被gc跟踪的对象的数目；
`objgraph.show_most_common_types([limit=10, objects])`表示显示常用类型对应的对象的数目。

- 定位和过滤对象。如`objgraph.by_type(typename[, objects])`表示根据传入的参数显示被gc跟踪的对象信息；`objgraph.at(addr)`表示根据给定的地址返回对象。

- 遍历和显示对象图。如`objgraph.show_refs(objs[, max_depth=3, extra_ignore=(), filter=None, too_many=10, highlight=None, filename=None, extra_info=None, refcounts=False])`表示从对象objs开始显示对象引用关系图；`objgraph.show_backrefs(objs[, max_depth=3, extra_ignore=(), filter=None, too_many=10, highlight=None, filename=None, extra_info=None, refcounts=False])`表示显示以objs的引用作为结束的对象关系图。

更多关于objgraph使用的API文档参见<http://mg.pov.lt/objgraph/objgraph.html>。下面来看使用objgraph的两个简单的例子。其中第一个例子生成对象的引用关系图，第二个显示不同类型对象的数目。

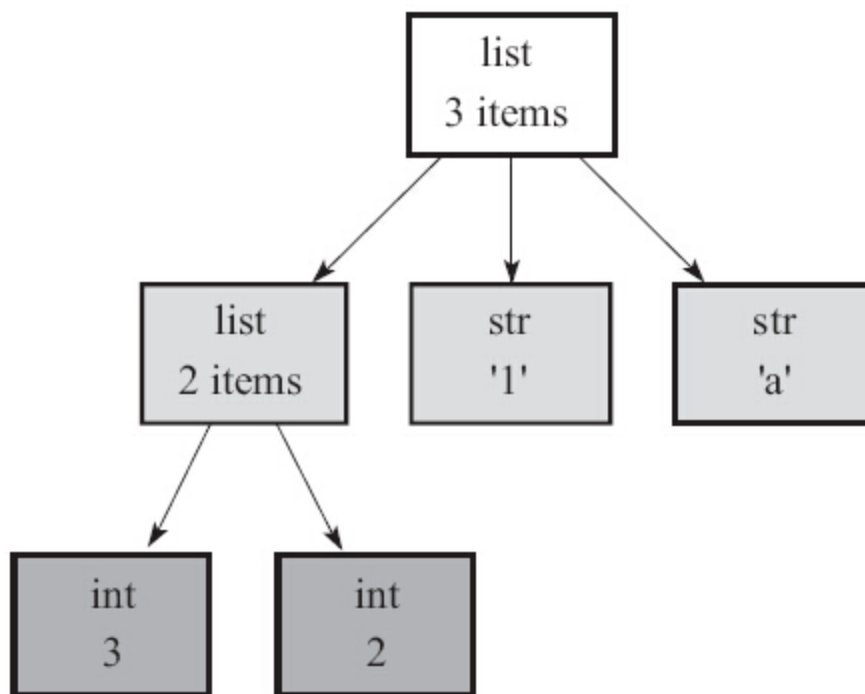


图8-3 对象x的引用关系图

1) 生成对象x的引用关系图。生成的关系图如图8-3所示。具体代码如下:

```
>>> import objgraph
>>> x = ['a', '1', [2, 3]]
>>> objgraph.show_refs([x], filename='test.png')
```

2) 显示常用类型不同类型对象的数目，限制输出前3行。代码如下:

```
>>> objgraph.show_most_common_types(limit = 3)
wrapper_descriptor      1031
```

function	975
builtin_function_or_method	615

建议83：努力降低算法复杂度

同一问题可用不同算法解决，而一个算法的优劣将直接影响程序的效率和性能。算法的评价主要从时间复杂度和空间复杂度来考虑。空间复杂度的分析相对来说要简单，并且在当前的计算硬件资源发展形势下，对空间复杂度的关注远没有时间复杂度高。因此降低算法的复杂度主要集中在对其时间复杂度的考量，本章侧重考虑时间复杂度。算法的时间复杂度是指算法需要消耗的时间资源，常使用大写字母O表示。如插入排序的时间复杂度为 $O(n^2)$ ，快速排序的最坏运行时间是 $O(n^2)$ ，但是平均运行时间则是 $O(n \log n)$ 。同一算法对应的不同代码实现的性能差异可能仅仅体现在其系数上，但数量级上仍然在同一水平，但不同时间复杂度的算法随着计算规模的扩大带来的性能差别则较为明显。下面是算法时间复杂度大O的排序比较：

$$O(1) < O(\log^* n) < O(n) < O(n \log n) < O(n^2) < O(c^n) < O(n!) < O(n^n)$$

因此对算法改进的目的是尽量往时间复杂度较低的O靠近。要降低算法的复杂度，首先要对算法复杂度进行分析。算法分析建立在一定的假设前提上：即一台给定的计算机执行每一条指令的时间是确定的，因此，对于获取字典中某个key对应的值，其时间复杂度为 $O(1)$ ，查找列表中某个元素，其时间复杂度最优为 $O(1)$ ，最坏的情况为 $O(n)$ 。下面的示例中用于求两个列表交集，即使函数中存在条件分支，虽然if部分运算的时间复杂度为 $O(1)$ ，但else部分需要循环遍历两个列表，其时间复杂度为 $O(n^2)$ ，因此最终的算法复杂度为 $O(n^2)$ 。

```
def intersection1(list1, list2):
    result = {}
    if len(list1) < 5:                                     #
        print list1                                       #
    else:
        #
        #
```

时间复杂度为 $O(1)$

时间复杂度为 $O(n^2)$

```

for item in list1:
    if item in list2:
        result[item] = True
return result.keys()

```

需要特别说明，算法的复杂度分析的粒度非常重要，其前提一定是粒度相同的指令执行时间近似，千万不能将任意一行代码直接当做 $O(1)$ 进行分析。例如上面的例子中如果有其他函数再调用`intersection1`，纵然在调用函数中只有一行代码，该行代码的时间复杂度仍然要按照 $O(n^2)$ 计算。另外，算法复杂度分析建立在同一级别语言实现的基础上，如果Python代码中含有C实现的代码，千万不能将两者混在一起进行评估。关于更多算法分析的思想和方法，读者可以查看数据结构与算法相关资料。

Python常见数据结构基本操作的时间复杂度如表8-4所示。

表8-4 常见数据结构基本操作的时间复杂度

数据结构	操 作	平均时间复杂度	最差时间复杂度
list	复制	$O(n)$	$O(n)$
	追加、取元素的值，给某个元素赋值	$O(1)$	$O(1)$
	插入、删除某个元素，迭代操作	$O(n)$	$O(n)$
	切片操作	$O(k)$	$O(k)$
set	$x \text{ in } s$	$O(1)$	$O(n)$
	并 $s t$	$O(\text{len}(s)+\text{len}(t))$	
	交 $s\&t$	$O(\min(\text{len}(s), \text{len}(t)))$	$O(\text{len}(s) * \text{len}(t))$
	差 $s-t$	$O(\text{len}(s))$	
dict	获取修改元素的值，删除	$O(1)$	$O(n)$
	迭代操作	$O(n)$	$O(n)$
collections.deque	入列、出列（包括左边出入列）	$O(1)$	$O(1)$
	扩大队列	$O(k)$	$O(k)$
	删除元素	$O(n)$	$O(n)$

建议84：掌握循环优化的基本技巧

循环的优化应遵循的原则是尽量减少循环过程中的计算量，多重循环的情形下尽量将内层的计算提到上一层。

1) 减少循环内部的计算。下面两个示例实现的是同一功能，但提倡使用第二种循环实现，因为第一种循环中`d=math.sqrt(y)`位于循环内部，每次循环过程中都会重新计算一遍，无形中增加了系统开销。测试结果表明，第二种运算的计算速率比第一种运算的速率快40%~60%。

示例一：

```
for i in range(iter):
    d=math.sqrt(y)
    j+=i*d
```

示例二：

```
d=math.sqrt(y)
for i in range(iter):
    j+=i*d
```

2) 将显式循环改为隐式循环。假设求等差数列1, 2....., n的和，可以直接通过如下循环来计算：

```
sum = 0
for i in xrange(n+1):
    sum = sum+i
```

也可以直接写出得到计算结果的值： $n*(n+1)/2$ 。显然直接计算表达式的值效率更高，程序中如果有类似的情形，可以将显式循环改为隐式。当然这可能会带来另一个负面影响：牺牲了代码的可读性。因此这种情况下清晰、恰当的注释是非常必要的。

3) 在循环中尽量引用局部变量。在命名空间中局部变量优先搜索，因此局部变量的查询会比全局变量要快，当在循环中需要多次引用某一个变量的时候，尽量将其转换为局部变量。下面的例子中如果使用示例二代替示例一，性能将提高10%~15%。

示例一：

```
x = [10, 34, 56, 78]
def f(x):
    for i in xrange(len(x)):
        x[i] = math.sin(x[i])
    return x
```

示例二：

```
def g(x):
    loc_sin = math.sin
    for i in xrange(len(x)):
        x[i] = loc_sin(x[i])
    return x
```

4) 关注内层嵌套循环。在多层嵌套循环中，重点关注内层嵌套循环，尽量将内层循环的计算往上层移。如下面的示例一中，`v1[i]`在第二层循环`for j in range(len(v2))`时针对每个*i*其值保持不变，因此可以在外层循环中使用临时变量替代而不是每次都重新计算，如示例二所示。

示例一：

```
for i in range(len(v1)):
    for j in range(len(v2)):
        x = v1[i] + v2[j]
```

示例二:

```
for i in range(len(v1)):
    v1i = v1[i]
    for j in range(len(v2)):
        x = v1i + v2[j]
```

建议85：使用生成器提高效率

斐波那契数列相信大家都不陌生，这是常见的编程题目，也是很多书籍中喜欢引用的例子。我们这里也不免落于俗套，就以这个例子开场吧。斐波那契是一个简单的递归数列，数列满足这样的规律：除了前两个数，任何其他数都可以由其前面两个数相加得到，即 $f(N)=f(N-1)+f(N-2)$ ， $n>2$ 。Python中有多种方法实现这个数列，我们来看其中的一种。

```
>>> def fab(n):
...     i,a,b = 0,0,1
...     foblist = []
...     while i < n:
...         foblist.append(b)
...         a,b = b,a+b
不借助中间变量交换两个变量的方法
...         i = i+1
...     return foblist
...
>>> print fab(4)
[1, 1, 2, 3]
```

想一想，上面的例子有没有更好的实现方法呢？显然有！在介绍具体实现之前我们先来了解生成器的有关知识。

生成器的语法在Python2.2中就引入了，但实际应用过程中还是有人不会选择使用它，特别是有过其他语言基础的，主要是思维上难以转换过来。生成器的概念其实非常简单，如果一个函数体中包含有yield语句，则称为生成器（generator），它是一种特殊的迭代器

（iterator），也可以称为可迭代对象（iterable）。可迭代对象、迭代器、生成器这三者之间的关系可以简单地表示成如图8-4所示的形式。

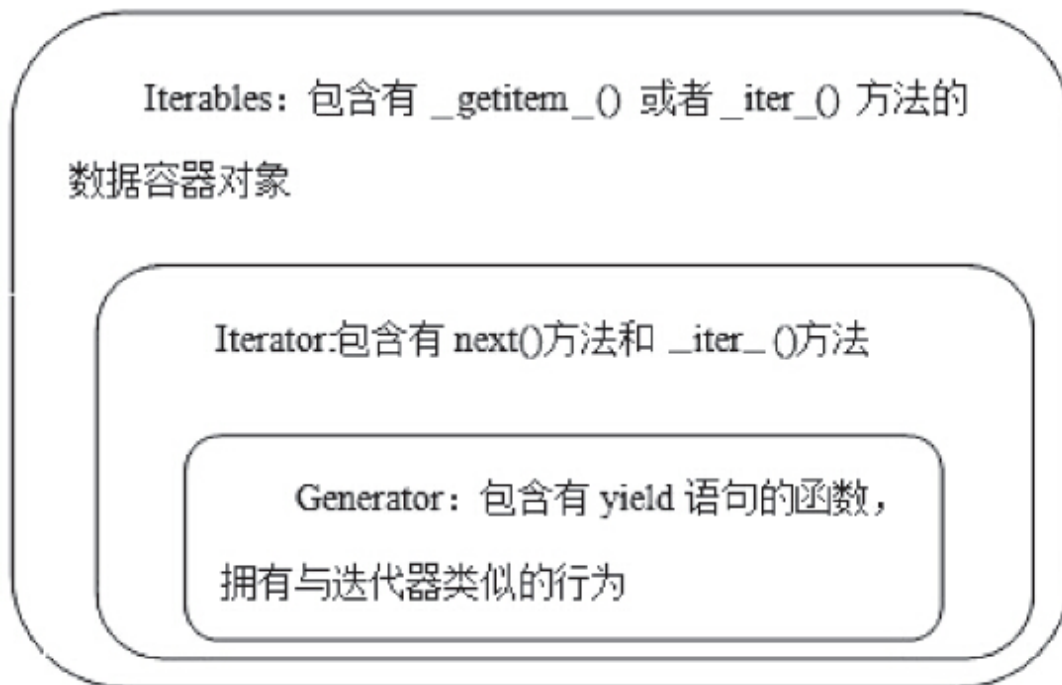


图8-4 可迭代对象、迭代器、生成器三者之间的关系

对生成器的调用会返回一个迭代器，使用`next()`方法可以获取下一个元素或者抛出`StopIteration`异常。

```
>>> def mygen(x):
...     for i in range(x):
...         yield i
...
>>> d = mygen(1)
>>> d                                     #
生成器对象，拥有iter()
和next()
方法
<generator object mygen at 0x005162B0>
>>> d.next()
0
>>> d.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

实际上当需要在循环过程中依次处理一个序列中的元素的时候，就应该考虑生成器。当然，要深入理解生成器，必须透彻理解`yield`语句。`yield`语句与`return`语句相似，当解释器执行遇到`yield`的时候，函数

会自动返回yield语句之后的表达式的值。不过与return不同的是，yield语句在返回的同时会保存所有的局部变量以及现场信息，以便在迭代器调用next()或者send()方法的时候还原，而不是直接交给垃圾回收器（return()方法返回后这些信息会被垃圾回收器处理）。这样就能够保证对生成器的每一次迭代都会返回一个元素，而不是一次性在内存中生成所有的元素。自Python2.5开始，yield语句变为表达式，可以直接将其值赋给其他变量，如x=(yield y)。结合一个例子来看yield语句在生成器函数调用的时候执行状态，下面的函数代表数列1，-3，5，-7，9，.....。

```
>>> def series():
...     print "begin:"
...     m=1.0; n = 1
...     print "while begin"
①
...     while(1):
②
...         print "yield a data"
...         yield m/n
③
...         m = m+2
④
...         n = n* -1
...         print "end"
...
>>>
```

上面的代码用状态机可以表示为如图8-5所示的形式。状态机中状态1表示从函数定义到标注1处的所有语句的集合，状态2表示标注2的语句，状态3表示标注2到3之间的所有语句的集合，状态4表示从标注4后到结束的语句。

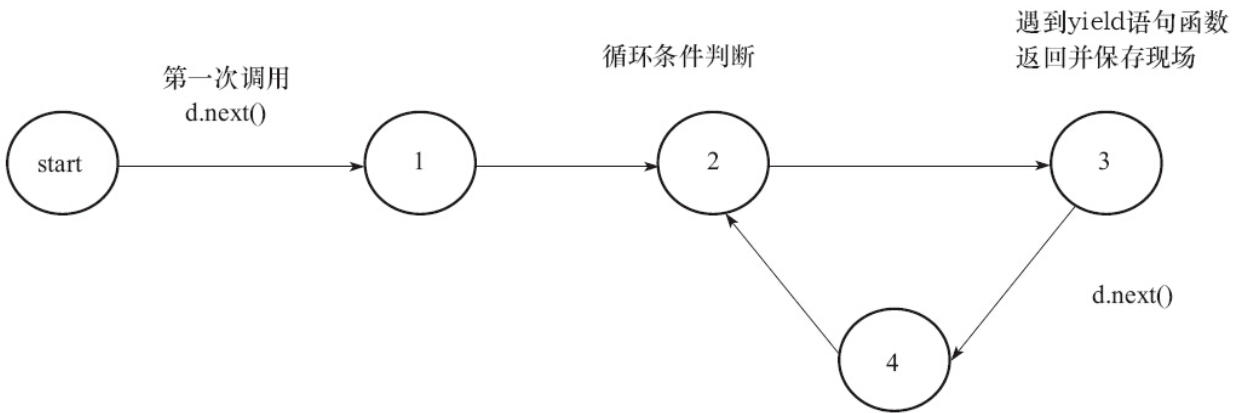


图8-5 示例程序的状态机形式的表示

运行上面的程序你会惊讶地发现，即使while中的条件永远为真，代码也不会陷入无限循环的状态，而是每调用一次next()方法产生一个数，这样生成器的优势就体现出来了。

```

>>> d = series()
>>> d.next()           #
第一次调用next
执行到状态2
， 循环条件为1
， 转到状态3
， 遇到yield
语句返回对
应的值并保留现场
begin:
while begin
yield a data
1.0
>>> d.next()           #
之后每次调用d.next()
从状态3
开始转向状态4
， 也就是yield
语句之后的第一句
语句开始执行， 4
直接转到2
， 循环条件永远为真， 从而再次转到状态3
end
yield a data
-3.0
  
```

生成器的优点总体来说有如下几条：

·生成器提供了一种更为便利的产生迭代器的方式，用户一般不需要自己实现__iter__和next方法，它默认返回一个迭代器。

·代码更为简洁、优雅。

·充分利用了延迟评估（**Lazy evaluation**）的特性，仅在需要的时候才产生对应的元素，而不是一次生成所有的元素，从而节省了内存空间，提高了效率，理论上无限循环成为可能而不会导致**MemoryError**，这在大数据处理的情形下尤为重要。

·使得协同程序更为容易实现。协同程序是有多个进入点，可以挂起恢复的函数，这基本就是**yield**的工作方式。Python2.5之后生成器的功能更加完善，加入了**send()**、**close()**和**throw()**方法。其中**send()**不仅可以传递值给**yield**语句，而且能够恢复生成器，因此生成器能大大简化协同程序的实现。

现在我们回过头来看看本节开头的例子，使用生成器来实现是不是更为简洁呢？

```
>>> def fib(n):
...     a = b = 1
...     for i in range(n):
...         yield a
...         a, b = b, a+b
... 
```

建议86：使用不同的数据结构优化性能

在解决性能问题的时候，如果已经到了非改代码不可的情况，考虑到Python中的查找、排序常用算法都已经优化到极点（虽然对`sort()`使用`key`参数比使用`cmp`参数有更高的性能仍然值得一提），那么首先应当想到的是使用不同的数据结构优化性能。

首先来看最常用的数据结构——`list`，它的内存管理类似C++的`std::vector`，即先预分配一定数量的“车位”，当预分配的内存用完时，又继续往里插入元素，就会启动新一轮的内存分配。`list`对象会根据内存增长算法申请一块更大的内存，然后将原有的所有元素拷贝过去，销毁之前的内存，再插入新元素。当删除元素时，也是类似，删除后发现已用空间比预分配空间的一半还少时，`list`会另外申请一块小内存，再做一次元素拷贝，然后销毁原有的大内存。可见，如果`list`对象经常有元素数量的“巨变”，比如膨胀、收缩得很频繁，那么应当考虑使用`deque`。

`deque`就是双端队列，同时具备栈和队列的特性，能够提供在两端插入和删除时复杂度为 $O(1)$ 的操作。相对于`list`，它最大的优势在于内存管理方面。如果不熟悉C++的`std::deque`，那么可以把`deque`想象为多个`list`连在一起（仅为比喻，非精确描述），“像火车一样，每一节车厢可以载客”，它的每一个“`list`”也可以存储多个元素。它的优势在插入时，已有空间已经用完，那么它会申请一个“车厢”来容纳新的元素，并将其与已有的其他“车厢”串接起来，从而避免元素拷贝；在删除元素时也类似，某个“车厢”空了，就“丢弃”掉，无需移动元素。所以当出现元素数量“巨变”时，它的性能比`list`要好上许多倍。

对于list这种序列容器来说，除了pop(0)和insert(0, v)这种插入操作非常耗时之外，查找一元素是否存在其中，也是O(n)的线性复杂度。在C语言中，标准库函数bsearch()能够通过二分查找算法在有序队列中快速查找是否存在某一元素。在Python中，对保持list对象有序以及在有序队列中查找元素有非常好的支持，这是通过标准库bisect来实现的。

bisect并没有实现一种新的“数据结构”，其实它是用来维护“有序列表”的一组函数，可以兼容所有能够随机存取的序列容器，比如list。它可使在有序列表中查找某一元素变得非常简单。

```
def index(a, x):
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError
```

保持列表有序需要付出额外的维护工作，但如果业务需要在元素较多的列表中频繁查找某些元素是否存在或者需要频繁地有序访问这些元素，使用bisect则相当值得。

对于序列容器，除了插入、删除、查找之外，还有一种很常见的需求是就获取其中的极大值或极小值元素，比如在查找最短路径的A*算法中就需要在Open表中快速找到预估值最小的元素。这时候，可以使用heapq模块。类似bisect，heapq也是维护列表的一组函数，其中最先接触的必然是heapify()，它的作用是把一个序列容器转化为一个堆。

```
>>> import heapq
>>> import random
>>> alist = [random.randint(0, 100) for i in xrange(10)]
>>> alist
[59, 62, 38, 18, 26, 92, 9, 57, 52, 97]
>>> heapq.heapify(alist)
>>> alist
[9, 18, 38, 52, 26, 92, 59, 57, 62, 97]
```

可以看到转化为堆后，`alist`的第一个元素`alist[0]`是整个列表中最小的元素，`heapq`将保证这一点，从而保证从列表中获取最小值元素的时间复杂度是 $O(1)$ 。

```
>>> heapq.heappop(alist)
9
>>> alist
[18, 26, 38, 52, 97, 92, 59, 57, 62]
```

除了通过`heapify()`函数将一个列表转换为堆之外，也可以通过`heappush()`、`heappop()`函数插入、删除元素，针对常见的先插入新元素再获取最小元素、先获取最小元素再插入新元素的需求，还有`heappushpop(heap, item)`和`heapreplace(heap,item)`函数可以快速完成。从上例可以看出，每次元素增减之后序列的变化都很大，可以想象维护堆结构需要付出许多额外计算，所以千万不要“提前优化”乱用`heapq`，以免带来性能问题。

除此之外，`heapq`还有3个通用函数值得介绍，其中`merge()`能够把多个有序列表归并为一个有序列表（返回迭代器，不占用内存），而`nlargest()`和`nsmallest()`类似于C++中的`std::nth_element()`，能够返回无序列表中最大或最小的 n 个元素，并且性能比`sorted(iterable,key=key)[:n]`要高。

除了对容器的操作可能会出现性能问题外，容器中存储的元素也有很大的优化空间，这是因为在很多业务中，容器存储的元素往往是同一类型的，比如都是整数，而且整数的取值范围也确定，那么就可以使用`array`优化程序性能。

`array`实例化的时候需要指定其存储的元素类型，如'`c`'，表示存储的每个人元素都相当于C语言中的`char`类型，占用内存大小为1字节。

```
>>> import array
>>> a = array.array('c', 'string')
>>> a
array('c', 'string')
>>> a[0] = 'c'
>>> print a
array('c', 'cstring')
```

从上例可以看出，`array`对象与`str`不同，它是可变对象，可以随意修改某一元素的值。不过它最大的优势在于更小的内容占用。

```
>>> import sys
>>> a
array('c', 'cstring')
>>> sys.getsizeof(a)
62L
>>> l = list('cstring')
>>> sys.getsizeof(l)
152
```

看，内存占用只有使用了`list`的40%左右，这个优化效果在元素数量巨大的时候会更加明显。此外，还有性能方面的提升。

```
>>> t = timeit.Timer(''.join(a), "a = list('cstring')")
>>> t.timeit()
0.21462702751159668
>>> t = timeit.Timer("a.tostring()", "import array;a=array.array('c','cstring')")
>>> t.timeit()
0.1419069766998291
```

从容器到字符串的转变可以看出`array`的性能提升是比较大的，但也不能认为`array`在什么方面都有更好的性能。

```
>>> t = timeit.Timer("a.reverse()", "import array;a=array.array('c', 'cstring')")
>>> t.timeit()
0.15056395530700684
>>> t = timeit.Timer("a.reverse()", "a = list('cstring')")
>>> t.timeit()
0.08988785743713379
```

看，在这里`list`的性能要好些。所以性能优化一定要根据`profiler`的剖析结果来进行，经验往往靠不住，这和“不要提前优化”一样是性能

优化的基本原则。

建议87：充分利用set的优势

假设有这么个需求，希望能找出两个不同的给定目录下相同的文件名。该怎么实现呢？一个可行的方法是列出两个目录下所有的文件名，然后逐一比较找出相同的项。实现代码如下：





```
import os
def ListFilename(dirname, filesuffix):
    filelist = []
    os.chdir(dirname)
    for files in os.listdir("."):
        if files.endswith(filesuffix):
            filelist.append(files) # 找出满足条件的后缀条件的文件加入到列表中
    return filelist
filelistA = ListFilename("C:\\", ".log")
filelistB = ListFilename("C:\\temp\\", ".log")
filelistA.sort() # 对列表进行排序
filelistB.sort()
samefilelist=[] # 用来存放相同文件的列表
for a in filelistA: # 对列表进行循环比较
    for b in filelistB:
        if a == b:
            samefilelist.append(a)
print samefilelist
```

示例程序选择的数据结构为列表，首先对列表进行排序，然后进行逐项比较，其算法的复杂度为 $O(m \times n)$ ，其中 m 、 n 分别为两个列表的长度。那么请读者思考一下，有没有更好的选择呢？我们来了解一下集合（set）的基本知识点。Python中集合是通过Hash算法实现的无序不重复的元素集。创建集合通过set()方法来实现。

```
>>> set("hello")
set(['h', 'e', 'l', 'o'])
>>> a = [1, 2, "34", (5, 6)]
>>> set(a) # 方便地将列表转换为set
set([(5, 6), 1, 2, '34'])
```

集合中常见的操作以及对应的时间复杂度如表8-5所示。

表8-5 集合常见操作及时间复杂度

操 作	说 明	图形表示	示 例	时间复杂度	
				平 均	最 差
<code>s.union(t)</code>	s 和 t 的 并 集, $s \cup t$		<pre>>>> s.union(t) set([1, 2, 3, 4, 5, 6])</pre>	$O(\text{len}(s) + \text{len}(t))$	
<code>s.intersection(t)</code>	s 和 t 的 交 集, $s \cap t$		<pre>>>> s.intersection(t) set([1, 2, 3, 4])</pre>	$O(\min(\text{len}(s), \text{len}(t)))$	$O(\text{len}(s) * \text{len}(t))$
<code>s.difference(t)</code>	s 和 t 的 差 集, $s-t$, 在 s 中 存 在 但 在 t 中 不 存 在 的 元 素 组 成 的 集 合		<pre>>>> s.difference(t) set([]) >>> t.difference(s) set([5, 6])</pre>	$O(\text{len}(s))$	
<code>s.symmetric_difference(t)</code>	$s \Delta t$, s 和 t 的 并 集 减 去 s 和 t 的 交 集		<pre>>>> s = set([1,2,3]) >>> t = set([3,4,5,6]) >>> s.symmetric_difference(t) set([1, 2, 4, 5, 6]) >>> s.union(t) set([1, 2, 3, 4, 5, 6]) >>> s.intersection(t) set([3]) >>> s.union(t)-s.intersection(t) set([1, 2, 4, 5, 6])</pre>	$O(\text{len}(s))$	$O(\text{len}(s) * \text{len}(t))$

对表8-5时间复杂度这列仔细分析会发现，集合操作的复杂度基本为 $O(n)$ ，最差的情况下时间复杂度才为 $O(n^2)$ 。回过头来看本节开头的例子，你是不是会有这么一个想法：如果能够将对列表的操作改为对集合的操作，性能将会明显提高？那么事实是不是如我们所料呢？我们先来基于一些基本操作测试一下这两种数据结构在性能上的表现。

1) 对list求相同的元素，set求并集。当元素规模为100的时候测试结果显示，list的耗时大约为set操作的15倍。

```
Python -m timeit -n 1000 "[x for x in xrange(100) if x in xrange(60, 100)]"
1000 loops, best of 3: 133 usec per loop
```

```
Python -m timeit -n 1000 "set(xrange(100)).intersection(xrange(60, 100))"
1000 loops, best of 3: 8.99 usec per loop
```

当元素规模为1000时的测试结果显示，list耗时大约为set操作的144倍，如表8-6所示。

```
Python -m timeit -n 1000 "[x for x in xrange(1000) if x in xrange(600, 1000)]"
1000 loops, best of 3: 9.93 msec per loop
Python -m timeit -n 1000 "set(xrange(1000)).intersection(xrange(600, 1000))"
1000 loops, best of 3: 68.9 usec per loop
```

表8-6 list和set在求相同元素操作时的性能比较

操 作	元素数目	时间 (usec)
list 求相同元素	100	133
	1000	9930
set 求交集	100	8.99
	1000	68.9

1) 向list和set中添加元素，当元素规模为100的时候，list的耗时为set的9倍。

```
Python -m timeit -s "testset=set()" -s "for x in xrange(100): testset.add(x)"
"for x in xrange(100): x in testset"
100000 loops, best of 3: 11.5 usec per loop
Python -m timeit -s "tmp = list()" -s "for x in xrange(100): tmp.append(x)"
"for x in xrange(100): x in tmp"
10000 loops, best of 3: 104 usec per loop
```

当元素规模为1000的时候，list的耗时约为set的89倍，如表8-7所示。

表8-7 往list和set中添加元素的情况下性能比较

操 作	元素数目	时间 (usec)
向 set 中添加元素	100	11.5
	1000	105
向 list 中添加元素	100	104
	1000	9410

```
Python -m timeit -s "testset=set()" -s "for x in xrange(1000): testset.  
    add(x)" "for x in xrange(1000): x in testset"  
10000 loops, best of 3: 105 usec per loop  
Python -m timeit -s "tmp = list()" -s "for x in xrange(1000): tmp.append(x)"  
    "for x in xrange(1000): x in tmp"  
100 loops, best of 3: 9.41 msec per loop
```

从表8-7所示的测试数据中可以看出，**set**在某些操作上明显比**list**更高效，实际上**set**的**union**、**intersection**、**difference**等操作要比**list**的迭代要快。因此如果涉及求**list**交集、并集或者差等问题可以转换为**set**来操作。因此本节最初的例子将**list**转换为**set**再求交集会更为简洁高效，可修改为`print set(filelistA)&set(filelist)`。修改后测试结果表明，即使规模在10左右，**set**的效率仍然比**list**高了将近3倍（读者可以自行验证）。

建议88：使用multiprocessing克服GIL的缺陷

众所周知，GIL的存在使得Python中的多线程无法充分利用多核的优势来提高性能，这个问题困扰着很多开发者，也使很多人备受打击。一些人甚至提出质疑要求移去GIL，但由于种种原因目前还没有明确的迹象表明GIL会在随后的版本中消失。为了能够充分利用多核优势，Python的专家们提供了另外一个解决方案：多进程。

Multiprocessing由此而生，它是Python中的多进程管理包，在Python2.6版本中引进的，主要用来帮助处理进程的创建以及它们之间的通信和相互协调。它主要解决了两个问题：一是尽量缩小平台之间的差异，提供高层次的API从而使得使用者忽略底层IPC的问题；二是提供对复杂对象的共享支持，支持本地和远程并发。

类Process是multiprocessing中较为重要的一个类，用于创建进程，其构造函数如下：

```
Process([group [, target [, name [, args [, kwargs]]]])
```

其中，参数target表示可调用对象；args表示调用对象的位置参数元组；kwargs表示调用对象的字典；name为进程的名称；group一般设置为None。该类提供的方法与属性基本上与threading.Thread类一致，包括is_alive()、join([timeout])、run()、start()、terminate()、daemon（要通过start()设置）、exitcode、name、pid等。

不同于线程，每个进程都有其独立的地址空间，进程间的数据空间也相互独立，因此进程之间数据的共享和传递不如线程来得方便。

庆幸的是multiprocessing模块中都提供了相应的机制：如进程间同步操作原语Lock、Event、Condition、Semaphore，传统的管道通信机制pipe以及队列Queue，用于共享资源的multiprocessing.Value和multiprocessing.Array以及Manager等。

Multiprocessing模块在使用上需要注意以下几个要点：

1) 进程之间的通信优先考虑Pipe和Queue，而不是Lock、Event、Condition、Semaphore等同步原语。进程中的类Queue使用pipe和一些locks、semaphores原语来实现，是进程安全的。该类的构造函数返回一个进程的共享队列，其支持的方法和线程中的Queue基本类似，除了方法task_done()和join()是在其子类JoinableQueue中实现的以外。需要注意的是，由于底层使用pipe来实现，使用Queue进行进程之间的通信的时候，传输的对象必须是可以序列化的，否则put操作会导致PicklingError。此外，为了提供put方法的超时控制，Queue并不是直接将对象写到管道中而是先写到一个本地的缓存中，再将其从缓存中放入pipe中，内部有个专门的线程feeder负责这项工作。由于feeder的存在，Queue还提供了以下特殊方法来处理进程退出时缓存中仍然存在数据的问题。

·close(): 表明不再存放数据到queue中。一旦所有缓冲的数据刷新到管道，后台线程将退出。

·join_thread(): 一般在close方法之后使用，它会阻止直到的后台线程退出，确保所有缓冲区中的数据已经刷新到管道中。

·cancel_join_thread(): 需要立即退出当前进程，而无需等待排队的数据刷新到底层的管道的时候可以使用该方法，表明无需阻止到后台线程的退出。

Multiprocessing中还有个SimpleQueue队列，它是实现了锁机制的pipe，内部去掉了buffer，但没有提供put和get的超时处理，两个动作都是阻塞的。

除了multiprocessing.Queue之外，另一种很重要的通信方式是multiprocessing.Pipe。它的构造函数为multiprocessing.Pipe([duplex])，其中duplex默认为True，表示为双向管道，否则为单向。它返回一个Connection对象的组（conn1,conn2），分别代表管道的两端。Pipe不支持进程安全，因此当有多个进程同时对管道的一端进行读操作或者写操作的时候可能会导致数据丢失或者损坏。因此在进程通信的时候，如果是超过2个以上进程，可以使用queue，但对于两个进程之间的通信而言Pipe性能更快。下面看一个示例：

```
from multiprocessing import Process, Pipe, Queue
import time
def reader_pipe(pipe):
    output_p, input_p = pipe #
    返回管道的两端
    input_p.close()
    while True:
        try:
            msg = output_p.recv() #
            从pipe
            中读取消息
        except EOFError:
            break
def writer_pipe(count, input_p): #
    写消息到管道中
    for i in xrange(0, count):
        input_p.send(i) #
    发送消息
def reader_queue(queue): #
    利用队列来发送消息
    while True:
        msg = queue.get() #
        从队列中获取元素
        if (msg == 'DONE'):
            break
def writer_queue(count, queue):
    for ii in xrange(0, count):
        queue.put(ii) #
    放入消息队列中
    queue.put('DONE')
if __name__ == '__main__':
    print "testing for pipe:"
    for count in [10**3, 10**4, 10**5]:
        output_p, input_p = Pipe()
        reader_p = Process(target=reader_pipe, args=((output_p,
        input_p),))
```

```

        reader_p.start()                                #
启动进程
        output_p.close()
        _start = time.time()
        writer_pipe(count, input_p)                    #
写消息到管道中
        input_p.close()
        reader_p.join()                                #
等待进程处理完毕
        print "Sending %s numbers to Pipe() took %s seconds" % (count,
            (time.time() - _start))
        print "testing for queue:"
        for count in [10**3, 10**4, 10**5]:
            queue = Queue()                              #
利用queue
进行通信
            reader_p = Process(target=reader_queue, args=((queue),))
            reader_p.daemon = True
            reader_p.start()
            _start = time.time()
            writer_queue(count, queue)                    #
写消息到queue
中
            reader_p.join()
            print "Sending %s numbers to Queue() took %s seconds" % (count,
                (time.time() - _start))
输出比较:
testing for pipe:
Sending 1000 numbers to Pipe() took 0.15299987793 seconds
Sending 10000 numbers to Pipe() took 0.384999990463 seconds
Sending 100000 numbers to Pipe() took 2.09099984169 seconds
testing for queue:
Sending 1000 numbers to Queue() took 0.169000148773 seconds
Sending 10000 numbers to Queue() took 0.555000066757 seconds
Sending 100000 numbers to Queue() took 3.0790002346 seconds

```

上面的代码分别用来测试两个多线程的情况下使用pipe和queue进行通信发送相同数据的时候的性能，其中与pipe相关的函数为reader_pipe()和writer_pipe()，而与queue相关的主要函数为writer_queue()和reader_queue()。从函数输出可以看出，pipe所消耗的时间较小，性能更好。

2) 尽量避免资源共享。相比于线程，进程之间资源共享的开销较大，因此要尽量避免资源共享。但如果不可避免，可以通过multiprocessing.Value和multiprocessing.Array或者multiprocessing.sharedctypes来实现内存共享，也可以通过服务器进程管理器Manager()来实现数据和状态的共享。这两种方式各有优势，总体来说共享内存的方式更快，效率更高，但服务器进程管理器

Manager()使用起来更加方便，并且支持本地和远程内存共享。我们通过几个例子来看一下各自使用需要注意的问题。

示例一：使用**Value**进行内存共享。

```
import time
from multiprocessing import Process, Value
def func(val):
    多个进程同时修改val
    for i in range(10):
        time.sleep(0.1)
        val.value += 1
if __name__ == '__main__':
    v = Value('i', 0)
    使用value
    来共享内存
    processList = [Process(target=func, args=(v,)) for i in range(10)]
    for p in processList: p.start()
    for p in processList: p.join()
print v.value
```

上面的程序输出是多少？100对吗？你可以运行看看。Python官方文档中有个容易让人迷惑的描述：在**Value**的构造函数 **multiprocessing.Value(typecode_or_type,*args[,lock])**中，如果lock的值为**True**会创建一个锁对象用于同步访问控制，该值默认为**True**。因此很多人会以为**Value**是进程安全的，实际上要真正控制同步访问，需要实现获取这个锁。因此上面的例子要保证每次运行都输出100，需要将函数**func**修改如下：

```
def func(val):
    for i in range(10):
        time.sleep(0.1)
        with val.get_lock():
            仍然需要使用get_lock
            方法来获取锁对象
            val.value += 1
```

示例二：使用**Manager**进行内存共享。

```
import multiprocessing
def f(ns):
    ns.x.append(1)
```

```

        ns.y.append('a')
if __name__ == '__main__':
    manager = multiprocessing.Manager()
    ns = manager.Namespace()
    ns.x = []                                #manager
    内部包括可变对象
    ns.y = []
    print 'before process operation:', ns
    p = multiprocessing.Process(target=f, args=(ns,))
    p.start()
    p.join()
    print 'after process operation', ns      #
    修改根本不会生效

```

本意是希望`x=[1]`，`y=['a']`，程序输出是不是期望的结果呢？答案是否定的。这又是为什么呢？这是因为`manager`对象仅能传播对一个可变对象本身所做的修改，如有一个`manager.list()`对象，管理列表本身的任何更改会传播到所有其他进程。但是，如果容器对象内部还包括可修改的对象，则内部可修改对象的任何更改都不会传播到其他进程。因此，正确的处理方式应该是下面这种形式：

```

import multiprocessing
def f(ns,x,y):
    x.append(1)
    y.append('a')
    ns.x= x                                #
    将可变对象也作为参数传入
    ns.y = y
if __name__ == '__main__':
    manager = multiprocessing.Manager()
    ns = manager.Namespace()
    ns.x = []
    ns.y = []
    print 'before process operation:', ns
    p = multiprocessing.Process(target=f, args=(ns,ns.x,ns.y,))
    p.start()
    p.join()
    print 'after process operation', ns

```

3) 注意平台之间的差异。由于Linux平台使用`fork()`函数来创建进程，因此父进程中所有的资源，如数据结构、打开的文件或者数据库的连接都会被子进程共享，而Windows平台中父子进程相对独立，因此为了更好地保持平台的兼容性，最好能够将相关资源对象作为子进程的构造函数的参数传递进去。因此要避免如下方式：

```

f = None
def child(f):
    # do something
if __name__ == '__main__':
    f = open(filename, mode)
    p = Process(target=child)
    p.start()
p.join()
而推荐使用如下方式:
def child(f):
    print f
if __name__ == '__main__':
    f = open(filename, mode)
    p = Process(target=child, args=(f,))          #
将资源对象作为构造函数参数传入
    p.start()
    p.join()

```

需要注意的是，Linux平台上multiprocessing的实现是基于C库中的fork()，所有子进程与父进程的数据是完全相同，因此父进程中所有的资源，如数据结构、打开的文件或者数据库的连接都会被子进程共享。但Windows平台上由于没有fork()函数，父子进程相对独立，因此为了保持平台的兼容性，最好在脚本中加上“if __name__ == “__main__” :”的判断。这样可以避免有可能出现的RuntimeError或者死锁。

4) 尽量避免使用terminate()方式终止进程，并且确保pool.map中传入的参数是可以序列化的。在下面的例子中，如果直接将一个方法作为参数传入map中，会抛出cPickle.PicklingError异常，这是因为函数和方法是不可序列化的。

```

def run(self):
    def f(x):
        return x*x
    p = Pool()
    return p.map(f, [1,2,3])          #
直接传入函数f
cl = calculate()
print cl.run()                      #
抛出cPickle.PicklingError
异常

```

一个可行的正确做法如下：

```
import multiprocessing
def unwrap_self_f(arg, **kwarg):
    return calculate.f(*arg, **kwarg)          #
返回一个对象
class calculate(object):
    def f(self,x):
        return x*x
    def run(self):
        p = multiprocessing.Pool()
        return p.map(unwrap_self_f, zip([self]*3,[1,2,3]))
if __name__ == "__main__":
    cl = calculate()
    print cl.run()
```

建议89：使用线程池提高效率

我们知道线程的生命周期分为5个状态：创建、就绪、运行、阻塞和终止。自线程创建到终止，线程便不断在运行、就绪和阻塞这3个状态之间转换直至销毁。而真正占有CPU的只有运行、创建和销毁这3个状态。一个线程的运行时间由此可以分为3部分：线程的启动时间

（ T_s ）、线程体的运行时间（ T_r ）以及线程的销毁时间（ T_d ）。在多线程处理的情景中，如果线程不能够被重用，就意味着每次创建都需要经过启动、销毁和运行这3个过程。这必然会增加系统的相应时间，降低效率。而线程体的运行时间 T_r 不可控制，在这种情况下如何提高线程运行的效率呢？线程池便是一个解决方案。

线程池的基本原理如图8-6所示，它通过将事先创建多个能够执行任务的线程放入池中，所需要执行的任务通常被安排在队列中。通常情况下，需要处理的任务比线程数目要多，线程执行完当前任务后，会从队列中取下一个任务，直到所有的任务已经完成。

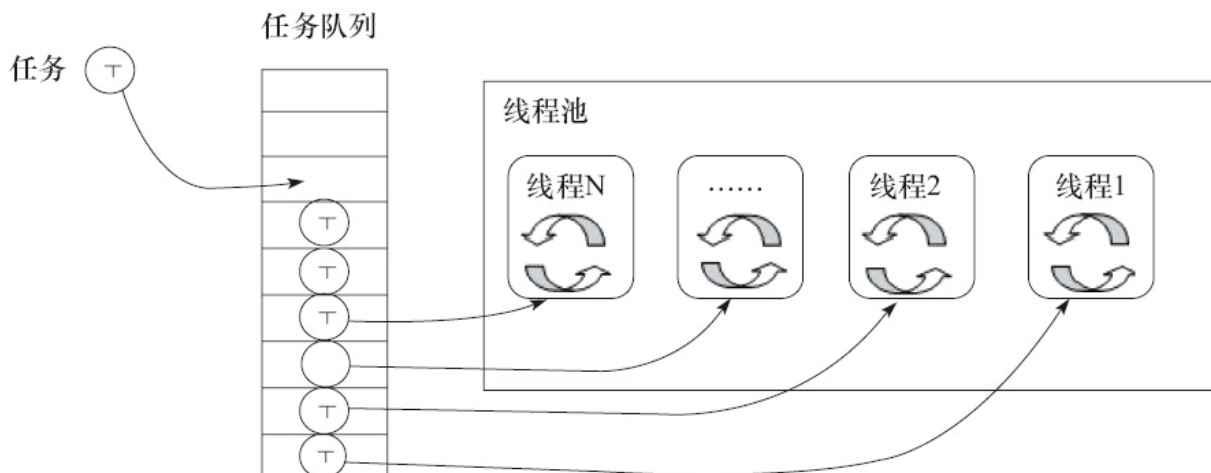


图8-6 线程池的基本原理

由于线程预先被创建并放入线程池中，同时处理完当前任务之后并不销毁而是被安排处理下一个任务，因此能够避免多次创建线程，从而节省线程创建和销毁的开销，带来更好的性能和系统稳定性。线程池技术适合处理突发性大量请求或者需要大量线程来完成任务、但任务实际处理时间较短的应用场景，它能有效避免由于系统中创建线程过多而导致的系统性能负荷过大、响应过慢等问题。

在Python中利用线程池有两种解决方案：一是自己实现线程池模式，二是使用线程池模块。我们先来看一个线程池模式的简单实现。

```
import Queue,sys,threading
import urllib2,os
#
处理request
的工作线程
class Worker(threading.Thread):
    def __init__( self, workQueue, resultQueue, **kws):
        threading.Thread.__init__( self, **kws )
        self.setDaemon( True )
        self.workQueue = workQueue
        self.resultQueue = resultQueue
    def run( self ):
        while True:
            try:
                callable, args, kws = self.workQueue.get(False)#
从队列中取出一个任务
                res = callable(*args, **kws)
                self.resultQueue.put( res ) #
存放处理结果到队列中
            except Queue.Empty:
                break
class WorkerManager: #
线程池管理器
    def __init__( self, num_of_workers=10):
        self.workQueue = Queue.Queue() #
请求队列
        self.resultQueue = Queue.Queue() #
输出结果的队列
        self.workers = []
        self._recruitThreads( num_of_workers )
    def _recruitThreads( self, num_of_workers ):
        for i in range( num_of_workers ):
            worker = Worker( self.workQueue, self.resultQueue )#
创建工作线程
            self.workers.append(worker) #
加入线程队列中
        def start(self): #
启动线程
            for w in self.workers:
                w.start()
            def wait_for_complete( self):
                while len(self.workers):
                    worker = self.workers.pop() #
从池中取出一个线程处理请求
```

```

        worker.join( )
        if worker.isAlive() and not self.workQueue.empty():
            self.workers.append( worker )           #
重新加入线程池中
    print "All jobs were completed."
    def add_job( self, callable, *args, **kwds ):
        self.workQueue.put( (callable, args, kwds) )           #
往工作队列中加入请求
    def get_result( self, *args, **kwds ):                 #
获取处理结果
        return self.resultQueue.get( *args, **kwds )
    def download_file(url):
        print "begin download",url
        urlhandler = urllib2.urlopen(url)
        fname = os.path.basename(url)+".html"
        with open(fname, "wb") as f:
            while True:
                chunk = urlhandler.read(1024)
                if not chunk: break
                f.write(chunk)
urls = ["http://wiki.python.org/moin/WebProgramming",
        "https://www.createspace.com/3611970",
        "http://wiki.python.org/moin/Documentation"
]
wm = WorkerManager(2)                                     #
创建线程池
for i in urls:
    wm.add_job( download_file, i )                       #
将所有请求加入队列中
wm.start()
wm.wait_for_complete()

```

自行实现线程池，需要定义一个**Worker**处理工作请求，定义**WorkerManage**来进行线程池的管理和创建，它包含一个工作请求列队和执行结果列队，具体的下载工作通过**download_file()**方法来实现。

相比自己实现的线程池模型，使用现成的线程池模块往往更简单。Python中线程池模块的下载地址为：
<https://pypi.python.org/pypi/threadpool>。该模块提供了以下基本类和方法。

1) **threadpool.ThreadPool**: 线程池类，主要的作用是用来分派任务请求和收集运行结果。主要有以下方法。

· **__init__(self, num_workers, q_size=0, resq_size=0, poll_timeout=5)**: 建立线程池，并启动对应 **num_workers** 的线程；**q_size**表示任务请求队列的大小，**resq_size**表示存放运行结果队列的大小。

·`createWorkers(self, num_workers, poll_timeout=5)`: 将`num_workers`数量对应的线程加入线程池中。

·`dismissWorkers(self, num_workers, do_join=False)`: 告诉`num_workers`数量的工作线程在执行完当前任务后退出。

·`joinAllDismissedWorkers(self)`: 在设置为退出的线程上执行`Thread.join`。

·`putRequest(self, request, block=True, timeout=None)`: 将工作请求放入队列中。

·`poll(self, block=False)`: 处理任务队列中新的请求。

·`wait(self)`: 阻塞用于等待所有执行结果。注意当所有执行结果返回后，线程池内部的线程并没有销毁，而在等待新的任务。因此，`wait()`之后仍然可以再次调用`pool.putRequests()`往其中添加任务。

2) `threadpool.WorkRequest`: 包含有具体执行方法的工作请求类。

3) `threadpool.WorkerThread`: 处理任务的工作线程，主要有`run()`方法以及`dismiss()`方法。

4)

`makeRequests(callable_,args_list,callback=None,exc_callback=_handle_thread_exception)`: 主要的函数，作用是创建具有相同的执行函数但参数不同的一系列工作请求。

最后看一个例子，将上一节多线程下载的例子改为用线程池来实现。

```
import urllib2
import os
```

```

import time
import threadpool
def download_file(url):
    print "begin download",url
    urlhandler = urllib2.urlopen(url)
    fname = os.path.basename(url)+".html"
    with open(fname, "wb") as f:
        while True:
            chunk = urlhandler.read(1024)
            if not chunk: break
            f.write(chunk)
urls = ["http://wiki.python.org/moin/WebProgramming",
        "https://www.createspace.com/3611970",
        "http://wiki.python.org/moin/Documentation"
]
pool_size = 2
pool = threadpool.ThreadPool(pool_size) #
创建线程池，大小为2
requests = threadpool.makeRequests(download_file, urls) #
创建工作请求
[pool.putRequest(req) for req in requests]
print "putting request to pool"
pool.putRequest(threadpool.WorkRequest(download_file,args=["http://chrisarndt.
de/projects/threadpool/api/"])) #
将具体的请求放入线程池
pool.putRequest(threadpool.WorkRequest(download_file,args=["https://pypi.python.
org/pypi/threadpool/"]))
pool.poll() #
处理任务队列中的新的请求
pool.wait()
print "destory all threads before exist"
pool.dismissWorkers(pool_size, do_join=True) #
完成后退出

```

建议90：使用C/C++模块扩展提高性能

Python具有良好的可扩展性，利用Python提供的API，如宏、类型、函数等，可以让Python方便地进行C/C++扩展，从而获得较优的执行性能。所有这些API却包含在Python.h的头文件中，在编写C代码的时候引入该头文件即可。来看一个简单的扩展例子。

1) 先用C实现相关函数：以实现素数判断为例，文件命名为testextend.c。也可以直接使用C语言实现相关函数功能后再使用Python进行包装。

```
include "Python.h"
static PyObject *pr_isprime(PyObject *self, PyObject *args){
    int n, num;
    if (!PyArg_ParseTuple(args, "i", &num))                #
        return NULL;
    if (num < 1) {
        return Py_BuildValue("i", 0);                        #C
    }
    n = num - 1;
    while (n > 1){
        if (num%n == 0) return Py_BuildValue("i", 0);
        n--;
    }
    return Py_BuildValue("i", 1);
}
static PyMethodDef PrMethods[] = {
    {"isPrime", pr_isprime, METH_VARARGS, "check if an input number is prime
    or not."},
    {NULL, NULL, 0, NULL}
};
void initpr(void){
    (void) Py_InitModule("pr", PrMethods);
}
```

上面的代码包含以下3部分。

·导出函数：C模块对外暴露的接口函数pr_isprime，带有self和args两个参数，其中参数args中包含了Python解释器要传递给C函数的所有

参数，通常使用函数PyArg_ParseTuple()来获得这些参数值。

·初始化函数：以便Python解释器能够对模块进行正确的初始化，初始化时要以init开头，如initp。

·方法列表：提供给外部的Python程序使用的一个C模块函数名称映射表 PrMethods。它是一个PyMethodDef结构体，其中成员依次表示方法名、导出函数、参数传递方式和方法描述。看下面这个例子。

```
struct PyMethodDef {
    char* ml_name;           #
    方法名
    PyCFunction ml_meth;     #
    导出函数
    int ml_flags;            #
    参数传递方法
    char* ml_doc;            #
    方法描述
};
```

参数传递方法一般设置为METH_VARARGS，如果想传入关键字参数，则可以将其与METH_KEYWORDS进行或运算。若不想接受任何参数，则可以将其设置为METH_NOARGS。该结构体必须以{NULL,NULL,0,NULL}所表示的一条空记录来结尾。

2) 编写setup.py脚本。

```
from distutils.core import setup, Extension
module = Extension('pr', sources = ['testextend.c'])
setup(name = 'Pr test', version = '1.0', ext_modules = [module])
```

3) 使用python setup.py build进行编译，系统会在当前目录下生成一个build子目录，里面包含pr.so和pr.o文件，如图8-7所示。

```
running build
running build_ext
building 'pr' extension
creating build
creating build/temp.linux-x86_64-2.4
gcc -pthread -fno-strict-aliasing -DNDEBUG -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstack-protector --param=ssp-buffer-size=4 -m64 -mtune=generic -D_GNU_SOURCE -fPIC -fPIC -I/usr/include/python2.4 -c testextend.c -o build/temp.linux-x86_64-2.4/testextend.o
creating build/lib.linux-x86_64-2.4
gcc -pthread -shared build/temp.linux-x86_64-2.4/testextend.o -o build/lib.linux-x86_64-2.4/pr.so
```

图8-7 使用Python进行编译

4) 将生成的文件pr.so复制到Python的site_packages目录下, 或者将pr.so所在目录的路径添加到sys.path中, 就可以使用C扩展的模块了, 如图8-8所示。

```
>>> import pr
>>> dir(pr)
['__doc__', '__file__', '__name__', 'isPrime']
>>> pr.isPrime(2)
1
```

图8-8 导入编译后的模块

更多关于C模块扩展的内容请读者参考<http://docs.python.org/2/c-api/index.html>。

建议91：使用 Cython 编写扩展模块

Python-API让大家可以方便地使用C/C++编写扩展模块，从而通过重写应用中的瓶颈代码获得性能提升。但是，这种方式仍然有几个问题让Pythonistas非常头疼：

1) 掌握C/C++编程语言、工具链有巨大的学习成本，如果没有这方面的技术积累，就无法快速编写代码，解决性能瓶颈。

2) 即便是C/C++熟手，重写代码也有非常多的工作，比如编写特定数据结构、算法的C/C++版本，费时费力还容易出错。

所以整个Python社区都在努力实现一个“编译器”，它可以把Python代码直接编译成等价的C/C++代码，从而获得性能提升。通过开发人员的艰苦工作，涌现出了一批这类工具，如Pyrex、Py2C和Cython等。而从Pyrex发展而来的Cython是其中的集大成者。

Cython通过给Python代码增加类型声明和直接调用C函数，使得从Python代码中转换的C代码能够有非常高的执行效率。它的优势在于它几乎支持全部Python特性，也就是说，基本上所有的Python代码都是有效的Cython代码，这使得将Cython技术引入项目的成本降到最低。除此之外，Cython支持使用decorator语法声明类型，甚至支持专门的类型声明文件，以使原有的Python代码能够继续保持独立，这些特性都使它得到广泛应用，如PyAMF、PyYAML等库都使用它编写自己的高效率版本。

安装Cython非常简单，使用pip能够很方便地安装。

```
pip install
-U cython
```

编译时间有点漫长，稍作等待，Cython就自动安装好了。然后我们可以尝试拿之前的`arithmetic.py`尝试一下，执行命令`cython arithmetic.py`，很快就完成了，但其实生成了一个`arithmetic.c`文件，它非常巨大，大概会有两三千行。是的，你没有看错，只有8行有效代码的`arithmetic.py`文件生成的C代码有两三千行。它的部分代码（`subtract`函数对应的代码的一分部）如下：

```
...
/* Python wrapper */
static PyObject *__pyx_pw_10arithmetic_7subtract(PyObject *__pyx_self, PyObject
*__pyx_args, PyObject *__pyx_kwds); /*proto*/
static PyMethodDef __pyx_mdef_10arithmetic_7subtract =
{__Pyx_NAMESTR("subtract"),
 (PyCFunction)__pyx_pw_10arithmetic_7subtract, METH_VARARGS|METH_KEYWORDS,
 __Pyx_DOCSTR(0)};
static PyObject *__pyx_pw_10arithmetic_7subtract(PyObject *__pyx_self, PyObject
*__pyx_args, PyObject *__pyx_kwds) {
    PyObject *__pyx_v_x = 0;
    PyObject *__pyx_v_y = 0;
    int __pyx_lineno = 0;
    const char *__pyx_filename = NULL;
    int __pyx_clineno = 0;
    PyObject *__pyx_r = 0;
    __Pyx_RefNannyDeclarations
    __Pyx_RefNannySetupContext("subtract (wrapper)", 0);
    {
        static PyObject **__pyx_pyargnames[] = {&__pyx_n_s_x, &__pyx_n_s_y, 0};
        PyObject* values[2] = {0, 0};
    }
    ...
}
```

看不懂？没有关系，机器生成的代码本来就不是为了给人看的，还是把它交给编译器吧。

```
$ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-aliasing \
-I/usr/include/python2.7 -o arithmetic.so arithmetic.c
```

又是一阵等待，编译、链接工作完成后，`arithmetic.so`文件就生成了。这时候可以像 `import` 普通的Python模块一样使用它。

```
$ python
>>> import arithmetic
>>> arithmetic.subtract(2, 1)
1
```

每一次都需要编译、等待未免麻烦，所以Cython很体贴地提供了无需显式编译的方案：pyximport。只要将原有的Python代码后缀名从.py改为.pyx即可。

```
$ cp arithmetic.py arithmetic.pyx
$ cd ~
$ python
>>> import arithmetic
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named arithmetic
>>> import pyximport; pyximport.install()
(None, <pyximport.pyximport.PyxImporter object at 0x10fbd05d0>)
>>> import arithmetic
>>> arithmetic.__file__
'/Users/apple/.pyxbld/lib.macosx-10.8-x86_64-2.7/arithmetic.so'
```

从__file__属性可以看出，这个.pyx文件已经被编译链接为共享库了，pyximport的确方便啊！掌握了Cython的基本使用方法之后，可以更进一步学习了。接下来要谈的是如何通过Cython把原有代码的性能提升许多倍，是的，Cython就是这么快！

在GIS中，经常需要计算地球表面上两点之间的距离。

```
import math
def great_circle(lon1,lat1,lon2,lat2):
    radius = 3956 #miles
    x = math.pi/180.0
    a = (90.0-lat1)*(x)
    b = (90.0-lat2)*(x)
    theta = (lon2-lon1)*(x)
    c = math.acos((math.cos(a)*math.cos(b)) +
    (math.sin(a)*math.sin(b)*math.cos(theta)))
    return radius*c
```

这段Python代码的执行效率可以通过timeit来确定。

```
import timeit
lon1, lat1, lon2, lat2 = -72.345, 34.323, -61.823, 54.826
num = 500000
t = timeit.Timer("p1.great_circle(%f,%f,%f,%f)" % (lon1,lat1,lon2,lat2),
                 "import p1")
print "Pure python function", t.timeit(num), "sec"
```

执行50万次大概需要：2.2秒，太慢了。接下来尝试使用Cython进行改写，为了避免一下子代码变化太大，只使用Cython的类型声明“技能”，看看能达到什么效果。

```
import math
def great_circle(float lon1,float lat1,float lon2,float lat2):
    cdef float radius = 3956.0
    cdef float pi = 3.14159265
    cdef float x = pi/180.0
    cdef float a,b,theta,c
    a = (90.0-lat1)*(x)
    b = (90.0-lat2)*(x)
    theta = (lon2-lon1)*(x)
    c = math.acos((math.cos(a)*math.cos(b)) +
    (math.sin(a)*math.sin(b)*math.cos(theta)))
    return radius*c
```

通过给great_circle函数的参数、中间变量增加类型声明，Cython代码看起来跟原有的Python代码并无很大不同，业务逻辑代码一行没改。使用timeit的测定结果是大概1.8秒，提速将近二成，说明类型声明对性能提升非常有帮助。这时候，还有一个性能瓶颈需要解决，那就是：调用的math库是一个Python库，性能较差。解决这个问题，需要用到Cython的另一个技能：直接调用C函数。

```
cdef extern from "math.h":
    float cosf(float theta)
    float sinf(float theta)
    float acosf(float theta)
def great_circle(float lon1,float lat1,float lon2,float lat2):
    cdef float radius = 3956.0
    cdef float pi = 3.14159265
    cdef float x = pi/180.0
    cdef float a,b,theta,c
    a = (90.0-lat1)*(x)
    b = (90.0-lat2)*(x)
    theta = (lon2-lon1)*(x)
    c = acosf((cosf(a)*cosf(b)) + (sinf(a)*sinf(b)*cosf(theta)))
    return radius*c
```

Cython使用`cdef extern from`语法，将`math.h`这个C语言库头文件里声明的`cosf`、`sinf`、`acosf`等函数导入代码中。因为减少了Python函数调用和调用时产生的类型转换开销，使用`timeit`测定这个版本的代码的效率仅需要大概0.4秒的时间，性能提升了5倍有余。

通过这个例子，可以掌握Cython的两大技能：类型声明和直接调用C函数。只要再进一步参考Cython的文档，就可以尝试在项目中使用了。比起直接使用C/C++编写扩展模块，使用Cython的方法方便得多，成本也更低。



注意

除了使用Cython编写扩展模块提升性能之外，Cython也可用来把之前编写的C/C++代码封装成`.so`模块给Python调用（类似`boost.python`/`SWIG`的功能），Cython社区已经开发了许多自动化工具。
